TECHNICAL SPECIFICATION

ISO/TS 10303-18

First edition
2021-05

Industrial automation systems and integration — Product data representation and exchange —

Part 18: Description methods: SysML XMI to Web services transformation

Reference number
ISO/TS 10303-18:2021(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 184, *Automation systems and integration*, Subcommittee SC 4, *Industrial data*.

A list of all parts in the ISO 10303 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product and independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

This document is a member of the description methods series. This document specifies the web services defined for ISO 10303. It includes the rationale for the selection of the web services and their scope. It does not include any specifications for management or maintenance of information and data on the server. This document supports the STEP extended architecture[12][13].

The need for standardized services

The traditional means for exchanging data using ISO 10303 is to share large packets of information using files. However, with increasingly interconnected organizations and tool sets, the need has arisen for secure flexible sharing of smaller packets of data. This need is valid for every industrial domain from analysis to design, and lifecycle support. It is also true for the different models of the STEP Extended architecture with services needed for the core, domain and data planning models.

The rapid evolution of web technologies means that the services definitions need to be independent of technology, with technology specific implementation schemas generated from the definitions.

The purpose of standardized services is to:

— define services against a model so that two or more vendors provide consistent implementations;

— provide a minimum specification to ensure consistency so that a third party can use the services from either of the vendors (such as “plug and play”).

The document development method:

Much of the content for Clause 5 was harvested from public deliverables of the Aerospace Technology Institute[21] APROCONE project[22] where intent was declared in these deliverables for the content to be used in this document. This was approved by all APROCONE project members.

The main components of this document are:

— types of services: a description of the categories of services from domain and technology independent to specific, and the rationale for the scope for this document;

— technology-independent services definition: the definition of the technology-independent services along with their inputs and outputs;

— technology-dependent methods: SysML XMI to OpenAPI 3.0.0 JSON schema;

— Annex B: SysML to OpenAPI - Canonical XMI and equivalent in OpenAPI JSON schema;

— Annex C: SysML to OpenAPI – Example files;

— Annex D: SysML to OpenAPI - A listing of, and mapping between, SDAI and HTTP response status codes.

Industrial automation systems and integration — Product data representation and exchange —

Part 18: Description methods: SysML XMI to Web services transformation

1 Scope

This document specifies the definition for services at the point of interaction between a client and server.

The following are within the scope of this document:

— the specification of the structure, components and conventions for domain- and technology-independent services implementation methods for STEP (ISO 10303-1);

— transformation of the SysML metamodel constructs to OpenAPI constructs for RESTful web services (see OpenAPI:3.0.0[25] and IETF RFC7231).

The following are outside the scope of this document:

— domain specific services definitions;

— the transformation of SysML metamodel constructs into OpenAPI constructs that are not used in the STEP extended architecture[12][13];

— the transformation of SysML metamodel constructs into OpenAPI constructs for other purposes than representing SysML constructs as STEP concepts;

— codes and scripts to transform SysML XMI to OpenAPI schema;

— the transformation of SysML constraints into OpenAPI schema;

— implementation of technology-specific services definitions other than RESTful OpenAPI;

— definition of management and maintenance of information and data on a server.

2 Normative references

There are no normative references in this document.

3 Terms, definitions, and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1.1 Terms and definitions for generic concepts

3.1.1.1
data
representation of information in a formal manner suitable for communication, interpretation, or processing by human beings or computers.

[SOURCE: ISO 10303-1:2021, 3.1.29]

3.1.1.2
implementation method
part of ISO 10303 that specifies a technique used by computer systems to exchange product data

[SOURCE: ISO 10303-1:2021, 3.1.39, modified — In the definition, the text after "data" has been removed.]

3.1.1.3
information
facts, concepts or instructions.

[SOURCE: ISO 10303-1:2021, 3.1.41]

3.1.1.4
information model
conceptual model of product data

Note 1 to entry: In ISO 10303, an information model is based on the Object-relationship modeling technique that organizes the product data as represented in different system aspects.

Note 2 to entry: In ISO 10303 information models are may be developed using EXPRESS modeling language.

EXAMPLE Application resource model for ISO 10303-242 managed model-based 3D engineering.

[SOURCE: ISO 10303-1:2021, 3.1.42 modified — The example has been changed.]

3.1.2 Terms and definitions for STEP concepts

3.1.2.1
entity
class of information defined by common properties

[SOURCE: ISO 10303-11:2004, 3.3.6, modified — In the definition, the article "a" has been removed.]

3.1.2.2
value
unit of data

[SOURCE: ISO 10303-11:2004, 3.3.22, modified — In the definition, the article "a" has been removed.]

3.1.3 Terms and definitions for SysML constructs

3.1.3.1
association
association classifies a set of tuples representing links between typed model elements

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.2]

3.1.3.2
auxiliary
a stereotype applied to an abstract *block* (3.1.3.3) that has no properties

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.3]

3.1.3.3
block
modular construct used for defining an *entity* (3.1.2.1)

Note 1 to entry: Used for defining objects in *information models* (3.1.1.4) such as Application activity model concepts, Application Data Planning objects, Application Domain Model Business Objects, Core model objects and ARM in SysML Entities. They may include reference, part, and value properties; constraints. They can be specializations of other Blocks.

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.4]

3.1.3.4
composite aggregation
responsibility for the existence of composed object.

Note 1 to entry: If a composite object is deleted, all of its part instances that are objects are deleted with it

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.5]

3.1.3.5
directed association
association between a collection of source model elements and a collection of target model elements that is said to be directed from the source elements to the target elements

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.6]

3.1.3.6
enumeration
Value Type whose values are enumerated

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.7]

3.1.3.7
enumeration literal
named value for an *enumeration*

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.8]

3.1.3.8
data type
type whose instances are identified only by their value

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.9]

3.1.3.9
generalization
directed relationship between a more general supertype and a more specific subtype

Note 1 to entry: Each Generalization relates a specific Classifier to a more general Classifier. Given a Classifier, the transitive closure of its general Classifiers is often called its generalizations, and the transitive closure of its specific Classifiers is called its specializations. The immediate generalizations are also called the Classifier's subtype, and where the Classifier is a Class, its supertype.

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.10]

3.1.3.10
part property
property that specifies a part with strong ownership and coincidental lifetime of its containing *block* (3.1.3.3).

Note 1 to entry: It describes a local usage or a role of the typing Block in the context of the containing Block. Every Part Property has Composite Aggregation and is typed by a Block.

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.12]

3.1.3.11
primitive type
definition of a predefined DataType, without any substructure

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.11]

3.1.3.12
reference property
property that specifies a reference of its containing *block* (3.1.3.3) to another block

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.13]

3.1.3.13
stereotype
limited kind of metaclass that cannot be used by itself but must always be used in conjunction with one of the metaclasses it extends

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.14]

3.1.3.14
value property
property of a *block* (3.1.3.3) that is typed with a ValueType

[SOURCE: ISO/TS 10303-15:2021, 3.1.3.15]

3.1.4 Terms and definitions used in OpenAPI specification

3.1.4.1
openapi
<OpenAPI> semantic version number field of the OpenAPI specification version

Note 1 to entry: A required fixed field in an OpenAPI schema.

Note 2 to entry: The full definition is provided in OpenAPI specification.

3.1.4.2
info
<OpenAPI> metadata about the schema field

Note 1 to entry: A required fixed field in an OpenAPI schema.

Note 2 to entry: The full definition is provided in OpenAPI specification.

3.1.4.3
servers
<OpenAPI> connectivity information to one or more target servers field

Note 1 to entry: A fixed field in an OpenAPI schema.

Note 2 to entry: The full definition is provided in OpenAPI specification.

3.1.4.4
tags
<OpenAPI> metadata representing logical grouping field.

Note 1 to entry: The full definition is provided in OpenAPI specification.

3.1.4.5
paths
<OpenAPI> available relative paths to the individual endpoints field

Note 1 to entry: A required fixed field in an OpenAPI schema.

Note 2 to entry: The full definition is provided in OpenAPI specification.

3.1.4.6
components
<OpenAPI> container for holding various schemas

Note 1 to entry: The full definition is provided in OpenAPI specification.

3.1.4.7
response object
<OpenAPI> description of a single response from an operation

Note 1 to entry: A response object may include design-time, static links to operations based on the response.

Note 2 to entry: The full definition is provided in OpenAPI specification.

3.1.4.8
responses
<OpenAPI> reusable *response objects* (3.1.4.7) field

Note 1 to entry: A fixed field in the field *components* in an OpenAPI schema.

Note 2 to entry: The full definition is provided in OpenAPI specification.

3.1.4.9
schema object
<OpenAPI> input and output data types definition

Note 1 to entry: The full definition is provided in *OpenAPI specification.*

3.1.4.10
schemas
<OpenAPI> reusable *schema objects* (3.1.4.9)

Note 1 to entry: A fixed field in the field *components* (3.1.4.6) in an OpenAPI schema.

Note 2 to entry: The full definition is provided in OpenAPI specification.

3.1.5 Terms and definitions used in hypertext transfer protocol

3.1.5.1
resource
<HTTP> HTTP request target

Note 1 to entry: HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources. Each resource is identified by a uniform resource identifier (URI).

Note 2 to entry: The full definition is provided in RFC 7231:2014, 2.

3.1.5.2
representation
<HTTP> information reflecting a past, current, or desired state of a given *resource* (3.1.5.1).

Note 1 to entry: the representation is in a format that can be readily communicated via the protocol, and that consists of a set of representation metadata and a potentially unbounded stream of representation data

Note 2 to entry: The full definition is provided in RFC 7231:2014, 3.

3.1.5.3
response status code
<HTTP> result of the attempt to realise the request represented as a three-digit integer code

Note 1 to entry: only the response status codes starting with 2 (Successful), 4 (Client error) and 5 (Server error) are used.

Note 2 to entry: The full definition is provided in RFC 7231:2014, 6.

3.1.5.4
POST
method that requests that the target *resource* (3.1.5.1) process the *representation* (3.1.5.2) contained in the request

Note 1 to entry: The full definition is provided in RFC 7231:2014, 4.3.3.

3.1.5.5
GET
method that requests a *representation* (3.1.5.2) for the target *resource* (3.1.5.1) is transferred

Note 1 to entry: The full definition is provided in RFC 7231:2014, 4.3.1.

3.1.5.6
PATCH
method that requests that a set of changes described in the request entity be applied to the *resource* (3.1.5.1)

Note 1 to entry: The full definition is provided in RFC 6902:2013, 1.

3.1.5.7
PUT
method that requests that the target *resource* (3.1.5.1) state be created or replaced with the state defined by the *representation* (3.1.5.2) contained in the request

Note 1 to entry: The full definition is provided in RFC 7231:2014, 4.3.4.

3.2 Abbreviated terms

AP	application protocol
API	application programming interface
APROCONE	advanced product concept analysis environment
CRUD	create read update delete
HTTP	hypertext transfer protocol
ID	identifier
JSON	Java script object notation
OCL	object constraint language
OMG	object management group
REST	representational state transfer
SDAI	standard data access interface
SDL	schema definition language

SOAP	simple object access protocol
STEP	standard for the exchange of product model data
SysML	system modeling language
UID	unique identifier
UML	unified modeling language
URI	universal resource indicator
UUID	universal unique identifier
WSDL	web services description language
XMI	xml metadata interchange
XML	extensible markup language
XSD	xml schema definition

4 Domain-independent, technology-independent services

4.1 Domain- and technology-independent services overview (CRUD+Query)

Figure 1 below shows the classification of the services according to their domain and implementation technology independence. This clause concerns the domain and implementation technology independent services definitions, shown in the bottom left of Figure 1.

NOTE 1 The domain independent, technology specific services are covered in Clause 5. The domain-specific services are not considered in this document.

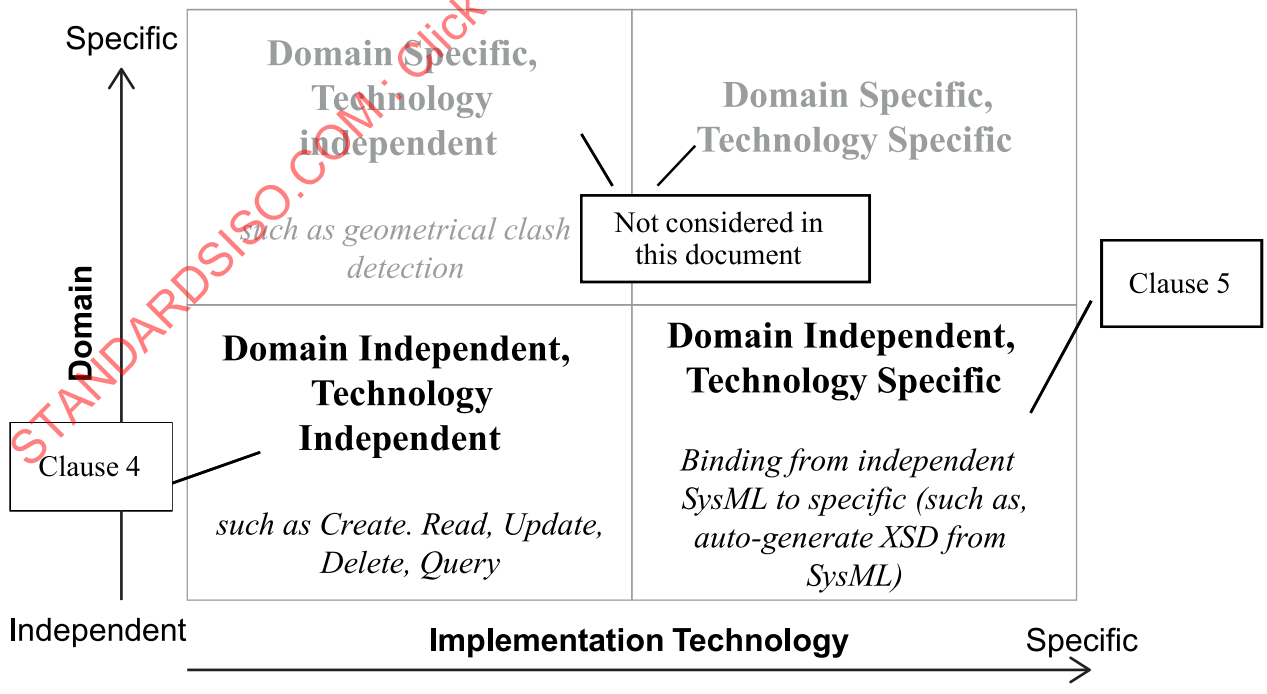

Figure 1 — Classification of services according to domain and technology independence

The domain- independent and technology-independent services are used to mutate (Create, Update, Delete) and interrogate (Read, Query) the data.

The standard to which the services apply, shall be directly reflected in any specializations and in the payload of the service request and response;

EXAMPLE 1 If the standard has an entity called “Circle” that has two part properties called “position” and “radius”, then the service can be CreateCircle(position, radius), or better still Create(type arguments) which for the Circle example equates to Create(type=Circle, arguments={position, radius}).

The services can also be used to define more advanced CRUD+Query services which combine aspects of the model. They are defined by modelling additional entities or properties and using parametric diagrams to detail what shall be created behind the scenes. The approach means that normal “CRUD+Query” can then be used for these new entities. Internal to an implementation, a developer can choose whether to directly use the new property or use the mapping defined in the parametric diagram, depending on which is best for their internal data model.

EXAMPLE 2 A Person “AssignTo” (see Figure 2) service can be defined to create/read the appropriate assignment object (OrganizationOrPersonInOrganizationAssignment or PersonInOrganization) depending on the type to which the person is being assigned.

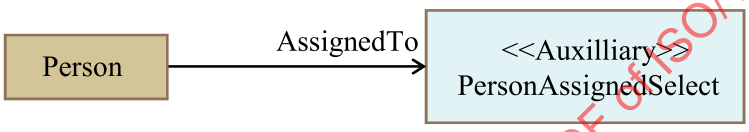

Figure 2 — Person AssignedTo

NOTE 2 The following types of services are not considered to be domain independent services and so are not considered in this document:

— combined services:
  — these services do complex tasks, but can be defined using a series of CRUD+Q services;
  — these types of services are generally specific to a particular use case and, as such, cannot be included in this document. Instead, they can be implemented in the client;
  — if a generic combined service is needed, then the approach of abstracting the service to a property and using parametric diagrams to define the behaviour (as described in this subclause) can be suitable for defining this type of service;
— value-added services:
  — these services do other tasks with the data;
  — the parametric diagrams approach (as described in this subclause) can, in some instances, be used to define this type of service.

EXAMPLE 3 Calculation or method services.

EXAMPLE 4 Create Circle(pt1, pt2, pt3). Assuming a standard "ABC" that defined the circle by using a centre point and radius, this service first needs to compute the centre point and radius from the three points before it can use these values to create a Circle following the standard "ABC".

EXAMPLE 5 Read CircleArea(). This service needs to read the circle retrieving the radius and then compute the area

EXAMPLE 6 Compare two sets of data (for validation – are they the same).

4.2 Technology independent services definition

4.2.1 General

The following is the technology independent definition for the services.

As described in Example 1 of 4.1, the service definition is independent of implementation technology, but also independent of the domain (AP) or layer of the architecture. To achieve this independence, instead of specific services for entities, such as “CreateCircle(position, radius)”, the service definitions are like “Create(representation)”, where the “representation” contains the entity type and entity properties. The same service definition is used for any domain (AP) and layer of the architecture, with only a schema for the “representation” being domain or layer dependent. The technology-independent schema is the SysML Canonical XMI, with technology specific schemas extracted from the SysML definitions.

EXAMPLE 1 Technology-specific schemas such as XSD for XML, JSON schema for JSON and SDL for GraphQL[23].

A “representation” shall include both part properties and reference properties. Therefore, the services definition also includes a definition for “reference”.

EXAMPLE 2 A technology-specific definition for a reference can be a URI.

The services definition contains definitions for entities representation and reference. These are explained in more detail in 4.2.2, followed by details for each of the mutation services (see 4.2.3) and interrogation services (see 4.2.4). They are summarized by Figure 3.

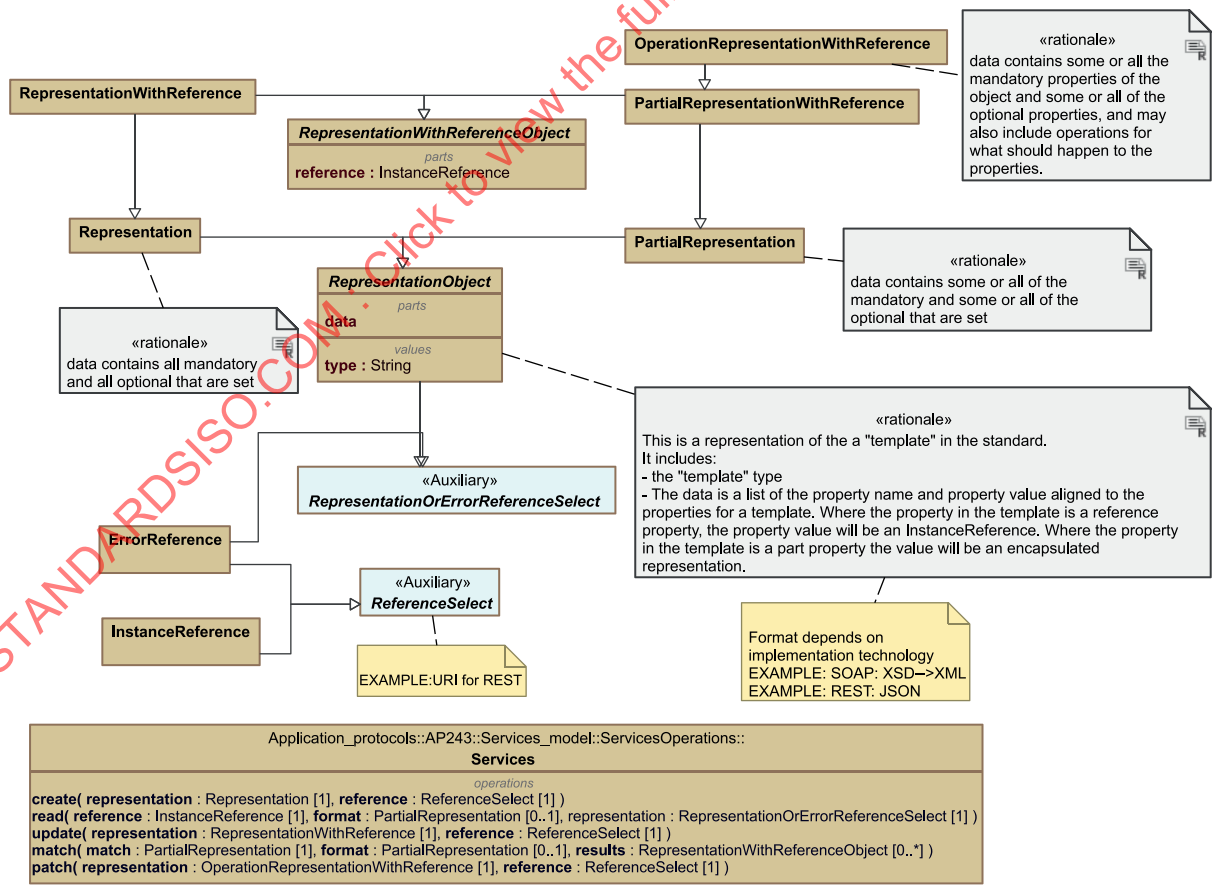

Figure 3 — Summary of services and their use of representation and reference entities

4.2.2 Representation, PartialRepresentation, OperationRepresentation and Reference

The Representation includes a string for the entity type and all public properties of the entity delineated as mandatory or optional as follows:

— reference properties (aggregation [None]) are defined as a reference;

NOTE 1 On the block definition diagrams, these have no diamond.

EXAMPLE 1 In Figure 4, ObjectWithPropertyDefinitions has optional multiple reference properties called “PropertyDefinitions” of type PropertyDefinition

— part and value properties (aggregation [Composite]) are defined as a Representation. As these are encapsulated this can lead to multiple levels of nested Representations.

NOTE 2 On the block definition diagrams, these are black diamonds;

EXAMPLE 2 In Figure 4, PropertyDefinition has a value property called “ValueType” of type ValueTypeEnum

EXAMPLE 3 In Figure 4, ObjectWithPropertyValues has optional multiple part properties called “Properties” of type PropertyValue.

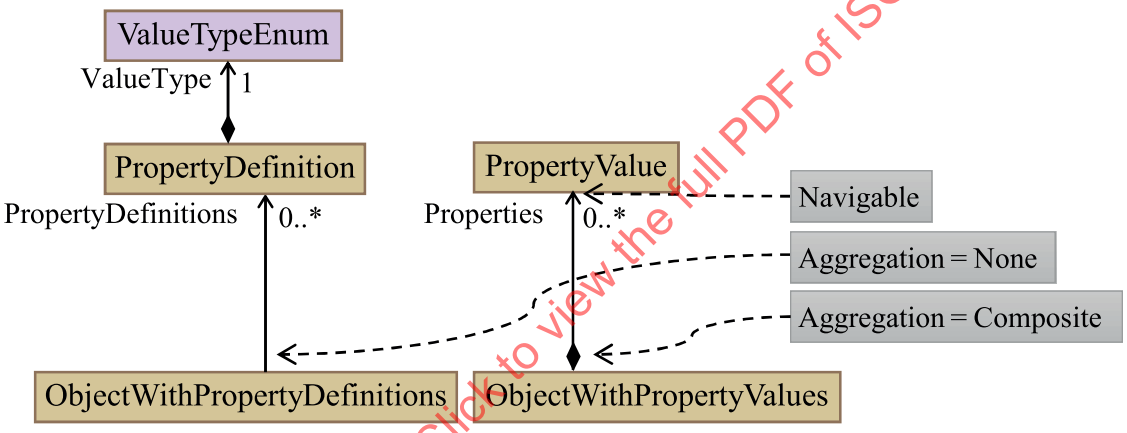

Figure 4 — Associations used in block definition diagrams

The PartialRepresentation is as the Representation, except none of the properties are mandatory.

The OperationRepresentation is as the PartialRepresentation, except it can also include operations for what shall happen to the properties in the Representation.

The InstanceReference is a reference to an object and the format or content is dependent on the implementation technology.

EXAMPLE 4 A Reference in a REST web service is a URI

EXAMPLE 5 A Reference in a SOAP web service can be an object with several properties as part of a WSDL.

The InstanceReference is returned by the mutate type of services (Create, Update, Delete). As well as a InstanceReference, the services can return an ErrorReference that is a reference a failure code for when the service does not execute as expected.

4.2.3 Mutation services

4.2.3.1 General

Mutation services are services that change the information stored in the server.

4.2.3.2 Create

4.2.3.2.1 Description

A mutation service that creates the specified object.

4.2.3.2.2 Definition

Send: a Representation (see 4.2.2) of the object defined by the object properties. This shall include all the mandatory properties as defined in the SysML model.

Behaviour: if a complete Representation is received with all the mandatory properties, create the object. If incomplete, return an error.

Response: Reference. This shall be a reference to the new object or a reference to a failure code.

4.2.3.2.3 Rationale

The rationale for mandating the send of all the mandatory properties is to avoid partially compliant implementations which are contradictory to the intent of standardization. If a server receives an incomplete Representation, it shall do nothing and respond with a suitable failure code reference.

The single Reference response is sufficient because:

— it can be assumed that, if response is an object reference, then the operation completed successfully and so an additional success/failure code is redundant;

— including the object Representation in the response is redundant because the client already has this information.

4.2.3.3 Update (overwrite)

4.2.3.3.1 Description

A mutation service that overwrites the specified object.

4.2.3.3.2 Definition

Send: InstanceReference and a Representation (see 4.2.2) of the object defined by the object properties. This shall include all the mandatory properties as defined in the SysML model.

Behaviour: it updates the entire object, overwriting all the properties, this includes un-setting any optional properties that are not included in the content, and replacing lists (not appending). It does not do any date or state checking.

Response: InstanceReference or ErrorReference. This shall be a reference to the updated object or a reference to a failure code.

4.2.3.3.3 Rationale

The behaviour of overwriting and un-setting optional properties not included, does not require any analysis on the part of the server so is the easiest to implement. It is what many web service implementations do today. As with the Create service, a response of an InstanceReference is all that is needed because the full representation is already available in the client.

4.2.3.4 Patch

4.2.3.4.1 Description

A mutation service that amends the properties for the object.

4.2.3.4.2 Definition

Send: InstanceReference and an OperationalRepresentation (see 4.2.2) of the object, defined by the object properties and optional associated patch operations.

Behaviour: amends the properties of the referenced object with the values included in the PartialRepresentation" according to the operation associated to each property, checking that the resultant object is valid. If the resultant object is not valid, a failure code ErrorReference shall be returned.

Response: InstanceReference or ErrorReference. This can be a reference to the amended object or a reference to a failure code.

4.2.3.4.3 Rationale

There are several possible modes of operation for patch, with different behaviour depending on the implementation technology. Therefore, the OperationalRepresentation is defined for each implementation technology.

EXAMPLE 1 Some implementation technologies assume operations for all properties in a request body, such that, if a property is missing from the request body, it is ignored.

EXAMPLE 2 Some technologies have a single operation in a request body that is applicable to all properties in the body, such as being able to declare if missing properties are ignored or deleted.

EXAMPLE 3 Some implementation technologies specify operations for each property in the request body.

4.2.4 Interrogation services

4.2.4.1 Description

Interrogation services are services that do not change the information stored in the server.

4.2.4.2 Read

4.2.4.2.1 Definition

Send: InstanceReference (see 4.2.2) to the object

Response: Representation (see 4.2.2) of the object defined by the object properties (all mandatory and any optionally defined properties) as defined in the SysML model.

4.2.4.2.2 Rationale

The rationale is to have the service as simple as possible to implement. Therefore, including a filter/format for the response has been discarded as the extra payload needed for the full Representation is not considered to outweigh the complexity. In addition, a filter/format is included in the Query service (see 4.2.4.3) which can be used in place of the read service if a filter/format was needed.

EXAMPLE Some implementation technologies such as GraphQL[23] eradicate the Read service altogether in favour of the Query service.

4.2.4.3 Query – equality pattern match

4.2.4.3.1 General

There are many types of query, but for this document an “equality pattern match” type of query is used.

4.2.4.3.2 Definition of equality pattern match

Send:

— match representation: a PartialRepresentation (see 4.2.2) of the object to match, containing the properties and references to match with the server data;

— format representation: a PartialRepresentation of what to return for the object, this defines which of the instance properties to include in the returned PartialRepresentations. If this is not provided, then it returns the full Representation.

Behaviour: this shall query the data base to find all the objects that have the properties and type equalling those in the match representation. For each of these found, it shall return a PartialRepresentation containing the properties defined in the format representation. The query is within a defined scope of the responsibility of the server; this can be within a server data repository or can expect a server to traverse repositories.

Response: list of PartialRepresentations, which can be an empty list.

4.2.4.3.3 Rationale for equality pattern match

The rationale for using an equality pattern match, as opposed to a query language, is to reduce the complexity of implementation and, thus, lower the barrier to conformance. This is achievable because the same representation is used for the mutation services and it allows implementers to reuse the mapping to their internal data.

The rationale for adopting a simple equality type of match, as opposed to more complex conditionals, was again to reduce the complexity. Multiple queries can be used to simulate the complex conditionals. The same rationale applies to the choice of single object type queries instead of expecting multiple object type queries. Multiple queries and client-side filtering can be used to achieve more tailored queries.

EXAMPLE Domain-specific match example is to find all geometrical entities within this volume for clash detection.

5 Technology-dependent methods

5.1 General

The following subclauses detail the transformation methods for converting the SysML Canonical XMI (as defined in ISO/IEC 19505-1, ISO/IEC 19514, ISO/IEC 19509 and W3C XML) into specific web services technology-dependent schemas. Each subclause defines the transformation method and Annex B explains the transformation with detailed examples.

Conformance to this document shall include satisfying the requirements of the web service implementation technology-dependent method(s) supported. An implementation shall support the following web service implementation technology dependent method: OpenAPI 3.0.0 JSON schema (see 5.3).

This document shall be unambiguously identified in an open information system by the code defined in Annex A.

5.2 Presentation conventions

For ease of identification, the fragments of JSON code are presented in a box. Fragments of code in the paragraphs use a fixed width font.

The items significant to support the explanations are often formatted using **bold** text effect to aid identification of the items in the code fragment. Angled brackets "<xxx>" are used to contain descriptive words of the content of the resulting JSON. Triple dots "..." are used to hide content not relevant to a fragment, these can contain a description of what is hidden using italic text effect.

EXAMPLE JSON fragment of presented in a box showing the use of **angled brackets and bold** text effect with triple dots that include an italic description.

```
GET https://<Server Address>/Organization/{uid}
...other content hidden...
```

5.3 SysML XMI to OpenAPI 3.0.0 JSON schema

5.3.1 General

The OpenAPI JSON schema format is defined in OpenAPI:3.0.0.[25] The schema defines the operations and payload for RESTful web services as defined in IETF RFC7231 and IETF RFC 6902. The format uses JSON as defined in ISO/IEC 21778.

OpenAPI JSON schemas are constructed as a series of nested JSON objects, called fields, held within a singular top-level schema object.

The fields used in this transformation are:

— openapi: OpenAPI specification version that the OpenAPI document uses;

— info: title description and version for the API;

— servers: connectivity information to target servers;

— tags: list of unique tags used for ordering the documentation;

— paths: the available paths and operations for the API;

— components:

  — responses: reusable response objects;

  — schemas: reusable schema objects.

Subclauses 5.3.2 to 5.3.7 define how each of the above fields is generated from the SysML Canonical XMI model of the information objects, including appropriate justifications and examples. Annex B explains the transformation using detailed fragments.

5.3.2 Field "openapi"

The field "openapi" is, by convention, the first field defined in an OpenAPI JSON schema but may be included later in the schema. The field states the OpenAPI Specification version used in the JSON schema and is mandatory.

EXAMPLE

```
"openapi": "3.0.0",
```


5.3.3 Field “info”

The field “info” contains fields for description, title and version of the JSON schema. The description shall include the document's N number. The title can contain free text. The version shall contain the version number following the specification of MAJOR.MINOR.PATCH[26].

EXAMPLE Field “info” for ISO 10303-4443, *OpenAPI JSON schema*:

```
"info": {
   "description": "ISO TC 184/SC 4/WG 12 N19999 ISO//TS 10303-4443 Edition 1: OpenAPI
specification for Modelling and Simulation information in a systems engineering
context.
OpenAPI 3.0 Web Services specification containing the services and object definitions.",
   "title": "ISO 10303-4443 OpenAPI3.0 Web Services specification",
   "version": "1.0.0"
},
```

5.3.4 Field “servers”

The field “servers” shall contain fields for the “description” and “url” for any available servers for evaluating the defined web services. The value of this field does not need to be specified.

EXAMPLE Field “servers”:

```
{
   "description": "API Auto Mocking Virtual Server",
   "url": "https://virtserver.tbd.com/ap243/1.0.0"
},
```

5.3.5 Field “tags”

The “tags” field is used to create groupings of similar web services, for ease of navigation in documentation, for the OpenAPI schema there shall be a tag for “Common” and a tag for each SysML non-abstract block.

EXAMPLE Field “tags” with the “Common” tag and a tag for a SysML non-abstract block name “NameOfBlock”:

```
{"name": "Common"},
{"name": "NameOfBlock"},
```

5.3.6 Field “paths”

5.3.6.1 General

The field “paths” shall contain the relative paths to the individual endpoints and their operations. A path is appended to the URL from the Server Object (see 5.3.4) to construct the full URL for the web services. With the exception of the MATCH service (see 5.3.6.6), all operations within the OpenAPI schema are called using unique endpoints; these declare the nature of the operation (such as POST, GET, PATCH, PUT) and the name of the entity to be handled (such as Organization). There shall be a single MATCH service endpoint that uses a POST operation, `POST /match`, that can be used for all entities represented in the SysML model by blocks that are not abstract.

All entities represented in the SysML model by blocks that are not abstract shall have individual `GET /<entity name>/{uid}`, `PATCH /<entity name>/{uid}` and `PUT /<entity name>/{uid}` operations.

Only entities represented in the SysML model by blocks that are not abstract and not encapsulated shall have individual `POST /<entity name>` operations. Encapsulated entities are created as part of the POST operation of the encapsulating entity. Therefore, they shall not have an individual POST operation.

Each operation entry shall have the following fields:

— “description”: a descriptive explanation of the services purpose and behaviour;

— “summary”: a brief summary of what the service does;

— “tags”: the tag where the service is to be grouped in the documentation;

— “operationId”: the unique identifier within the schema used to identify specific services within the API;

— “responses”: a list of possible server responses available on executing the given service.

PUT, PATCH and POST services have the additional field:

“requestBody”: the input JSON/XML attached to a service call, to provide the required output.

GET, PUT and PATCH services have the additional field:

“parameters”: the parameters available to identify objects held on the server.

5.3.6.2 POST

Every SysML block that is not abstract and not encapsulated shall have individual POST service specification.

POST services are assigned to URIs that represent names used in the SysML model, where the object type being created is the given name of the URI.

EXAMPLE 1 POST URI for SysML block named "Organization":

```
POST https://<Server Address>/Organization
```

For each POST service, a description, summary, tag and unique operationId shall be provided.

EXAMPLE 2 POST fields for the entity “Organization”:

```
"description": "Creates new 'Organization' objects",
"summary": "Create a new 'Organization' object",
"tags": [ "Organization" ]
"operationId": "post_Organization",
```

The “requestBody” field shall specify that it is required. The “content” field shall contain both JSON and XML schemas that reference the same components/schemas field entries (see 5.3.7.3).

EXAMPLE 3 POST requestBody for the entity “Organization”:

```
"requestBody": {
   "required": true,
   "content": {
      "application/json": {
           "schema": {"$ref": "#/components/schemas/Organization"}},
      "application/xml": {
          "schema": {"$ref": "#/components/schemas/Organization"}}}},
```

The “responses” field shall specify the permitted responses from the server. For all POST services, the server shall respond with one of the following HTTP codes (see 5.3.7.2):

— 201: Object created successfully;

— 400: Bad request;

— 401: Unauthorized;

— 403: Forbidden;

— 404: Not found.

All responses in POST services shall reference the response in the components/responses that is appropriate to the HTTP codes. The “responses” field is shown below.

```
"responses": {
   "201": {"$ref": "#/components/responses/201_POST"},
   "400": {"$ref": "#/components/responses/400"},
   "401": {"$ref": "#/components/responses/401"},
   "403": {"$ref": "#/components/responses/403"},
   "404": {"$ref": "#/components/responses/404"}
}
```

5.3.6.3 GET

Every SysML block that is not abstract shall have individual GET service specification.

GET services are assigned to URIs that represent names used in the SysML model, where the object type being created is the given name of the URI. These services also contain the parameter “uid” to return specific objects pertaining to that UID.

EXAMPLE 1 GET URI:

```
GET https://<Server Address>/Organization/{uid}
```

For each GET service, a description, summary, tag and unique operationId shall be provided.

EXAMPLE 2 GET description, summary, tag and operationId fields for the entity “Organization”:

```
"description": "Returns 'Organization' objects pertaining to a uid",
"summary": "Return 'Organization' object by uid",
"tags": [ "Organization" ]
"operationId": "get_Organization_uid",
```

The “parameters” field defines the expected input data when the service is called. The field shall contain the following fields:

— “description”: description of the parameter;

— “name”: name of the parameter defined in the service URI within curly brackets;

— “required”: boolean variable, declaring whether the parameter is required;

— “in”: declares where the parameter is defined;

— “schema”: reference to the schema component that defines a permitted input;

— “example”: example of a parameter input that complies with the schema. This field does not need to be specified.

All GET services in the OpenAPI schema shall contain the same “uri” parameter definition as shown below, where the “example” field does not need to be specified.

```
"parameters": [ {
   "description": "The uid of the object to be returned.",
   "name": "uid",
   "required": true,
   "in": "path",
   "schema": { "$ref": "#/components/schemas/ID" },
   "example": "_18_4_1_1b310459_1511445569604_398628_32624"
} ],
```

The “responses” field shall specify the permitted responses from the server. For all GET services, the server shall respond with one of five HTTP codes (see 5.3.7.2):

— 200: Object found;

— 400: Bad request;

— 401: Unauthorized;

— 403: Forbidden;

— 404: Not found.

All but the “200” response in GET services shall reference the response in the components/responses that is appropriate to the HTTP codes.

The “200” response is unique to each service and the “content” field shall contain both JSON and XML schemas that reference the components/schemas field entries for the object (see 5.3.7.3).

EXAMPLE 3 Responses for a GET service on the entity “Organization”:

```
"responses": {
   "200": {
      "content": {
         "application/json": {
            "schema": { "$ref": "#/components/schemas/Organization" } },
         "application/xml": {
            "schema": { "$ref": "#/components/schemas/Organization" } } },
      "description": "Resources read successfully" },
   "400": { "$ref": "#/components/responses/400" },
   "401": { "$ref": "#/components/responses/401" },
   "403": { "$ref": "#/components/responses/403" },
   "404": { "$ref": "#/components/responses/404" }
}
```

5.3.6.4 PATCH

Every SysML block that is not abstract shall have an individual PATCH service specification.

PATCH services are assigned to URIs that represent names used in the SysML model, where the object type being created is the given name of the URI. These services shall also contain the parameter “uid” to update specific objects pertaining to that UID.

EXAMPLE 1 PATCH URI:

```
PATCH https://<Server Address>/Organization/{uid}
```

For each PATCH service, a description, summary, tag and unique operationId shall be provided.

EXAMPLE 2 PATCH fields for the entity “Organization”:

```
"description": "Updates 'Organization' objects pertaining to a uid",
"summary": "Update 'Organization' object by uid",
"tags": [ "Organization" ]
"operationId": "patch_Organization_uid",
```

The “parameters” field of PATCH services is identical in definition to the corresponding field in GET (see 5.3.6.3).

The “requestBody” field follows IEFT RFC6902 and is not detailed in this document.

EXAMPLE 3 PATCH requestBody:

```
"requestBody": {
   "required": true,
   "content": {
    "application/json-patch+json": {
       "schema": {"type": "array", "items": {"type": "object"}}}}},
```

The responses field shall specify the permitted responses from the server. For all PATCH services, the server shall respond with one of five HTTP codes (see 5.3.7.2):

— 200: Object found;

— 400: Bad request;

— 401: Unauthorized;

— 403: Forbidden;

— 404: Not found.

All responses in PATCH services shall reference the response in the components/responses that is appropriate to the HTTP codes. The “responses” field is shown below.

```
"responses": {
   "200": { "$ref": "#/components/responses/200_PutPatch" },
   "400": { "$ref": "#/components/responses/400" },
   "401": { "$ref": "#/components/responses/401" },
   "403": { "$ref": "#/components/responses/403" },
   "404": { "$ref": "#/components/responses/404" }
}
```

5.3.6.5 PUT

Every SysML Block that is not abstract shall have an individual PUT service specification.

PUT services are assigned to URIs that represent names used in the SysML model, where the object type being created is the given name of the URI. These services shall also contain the parameter “uid” to replace specific objects pertaining to that UID.

EXAMPLE 1 PUT URI:

```
PUT https://<Server Address>/Organization/{uid}
```

For each PUT service, a description, summary, tag and unique operationId shall be provided.

EXAMPLE 2 Description, summary, tags and operationId fields for the entity “Organization”:

```
"description": "Replaces 'Organization' objects pertaining to a uid",
"summary": "Replace 'Organization' object by uid",
"tags": [ "Organization" ]
"operationId": "put_Organization_uid",
```

The “parameters” field of the PUT service shall be the same as the “parameters” field of the GET services (see 5.3.6.3).

The “requestBody” field of the PUT service shall be the same as the “requestBody” field of the POST service (see 5.3.6.2).

The “responses” field of PUT services shall be the same as the “responses” field of the PATCH service (see 5.3.6.4).

5.3.6.6 MATCH

The schema shall contain a single MATCH service specification. This service specification utilises a POST operation to a specific URI.

EXAMPLE 1 MATCH URI:

```
POST https://<Server Address>/match
```

The MATCH service shall have fields for description, summary, tag and operationId. The tag field shall contain “Common”, the operationId field shall contain “match”. The description and summary tags are free text.

EXAMPLE 2 MATCH service fields:

```
"description": " Equality matches content in payload and formats using.",
"summary": "Match payload and return using format.",
"tags": [ "Common" ]
"operationId": "match",
```

The “requestBody” field shall specify that it is required. The “content” field shall contain both JSON and XML schemas that reference the match_request in the components/schemas field entries (see 5.3.7.3). This requestBody “field” is shown below.

```
"requestBody": {
   "required": true,
   "content": {
      "application/json": {
           "schema": {"$ref": "#/components/schemas/match_request"}},
      "application/xml": {
           "schema": {"$ref": "#/components/schemas/match_request"}}}
},
```

The “responses” field shall specify the following HTTP codes (see 5.3.7.2):

— 200: Object found;

— 400: Bad request;

— 401: Unauthorized;

— 403: Forbidden;

— 404: Not found.

All but the “200” response in GET services shall reference the response in the components/responses that is appropriate to the HTTP codes.

The “200” response shall reference the “components/schemas/match_response”. This “responses” field is shown below.

```
"responses": {
   "200": {
      "content": {
         "application/json": {
           "schema": { "$ref": "#/components/schemas/match_response" } },
         "application/xml": {
           "schema": { "$ref": "#/components/schemas/match_response" } } },
      "description": "Matched Resources." },
   "400": { "$ref": "#/components/responses/400" },
   "401": { "$ref": "#/components/responses/401" },
   "403": { "$ref": "#/components/responses/403" },
   "404": { "$ref": "#/components/responses/404" } }
```

5.3.7 Field “components”

5.3.7.1 General

The “components” field contains the entities that are referenced throughout the OpenAPI schema. The components are split into two further fields:

— “responses”: reusable generic server responses;

— “schemas”: the definitions of entities used in the “requestBody” and “responseBody” fields of services (see 5.3.6).

5.3.7.2 Field “responses”

The responses field contains a set of reusable generic server responses. These are categorized according to their HTTP response status codes (as defined in IETF RFC7231), and either return a user message or an ID entity that conforms to a field in the “schemas” (see 6.3.7.3).

The following HTTP response status codes and corresponding fields are used to return an appropriate user message for their associated tasks (see 5.3.6).

— HTTP code 200:

  field “200_PutPatch”: used to confirm a resource has been updated.

— HTTP code 201:

  field “201_POST”: used to confirm a resource has been created.

— HTTP code 400:

  field “400”: used to indicate a bad request, usually caused by a syntax error.

— HTTP code 401:

  field “401”: used to indicate an unauthorized request, the user requires an authorization token.

— HTTP code 403:

  field “403”: used to indicate a forbidden request, denoting that the request was understood but the server is refusing to fulfil regardless of authorization.

— HTTP code 404:

  field “404”: used to indicate that the requested entity cannot be found.

POST services, (see 5.3.6.2) excluding MATCH, include an “HTTP code 201” response, given by the field “201_POST”. This response provides a response conforming to that of an ID entity schema, in both JSON and XML, similar to that used for “HTTP code 200” responses in GET services (see 5.3.6.3). This is defined below.

```
"201_POST": {
   "desciption": "Resource created successfully.",
   "content": {
      "application/json": {
         "schema": { "$ref": "#/components/schemas/ID" }},
      "application/xml": {
         "schema": { "$ref": "#/components/schemas/ID" }}}}
```

The complete “responses” field is defined below.

```
"responses": {
   "200_PutPatch": {"description": "Resource updated successfully."},
   "201_POST": {"description": "Resource created successfully.",
                "content": {
                    "application/json": {
                        "schema": {"$ref": "#/components/schemas/ID"}},
                    "application/xml": {
                        "schema": {"$ref": "#/components/schemas/ID"}}}},
  "400": {"description": "Bad Request."},
  "401": {"description": "Unauthorized." },
  "403": {"description": "Forbidden."},
  "404": {"description": "Not Found."}},
```

5.3.7.3 Field “schemas”

5.3.7.3.1 General

The “schemas” field defines the referenced schemas for the “requestBody” and “responseBody” for the services.

There is no inheritance in JSON schema. Therefore, all local and inherited properties are included in a schema entry. Subclauses 5.3.7.3.2 to 5.3.7.3.6 define how each piece of the SysML is transformed into the OpenAPI schema.

5.3.7.3.2 Blocks

Each non-abstract block shall have a “definition” in the “schemas” field using the name of the block.

The “definition” shall be of “type” “object” and shall have a single required property of the name of the block. This required property shall have the following:

— "type": "object";

— “properties”: {$href: {“$ref”: “#/components/schemas/uri"}, followed by the reference properties, part properties and value properties for the block

This required property can also have the following:

"description": The block description;

EXAMPLE 1 The start of the JSON Schema for a non-abstract block called “PropertyValue” is shown below, where the block properties are replaced by *“...italic text...”* in the example:

```
"PropertyValue": {
   "properties": {
      "PropertyValue": {
         "properties": {
            "$href": {"$ref": "#/components/schemas/uri"},
               ...properties for the block...
                     },
         "required": [...requried...],
         "type": "object"}},
    "required": ["PropertyValue"],
    "type": "object"},
```

Part properties are typed to blocks and those blocks can have subtypes that can be used in their place. Therefore, non-abstract blocks that are used for part properties and have subtypes shall have a “part definition” called {block name}Part that is “anyOf” the reference to the definition described above, and references to all the “part definition” for the immediate subtypes. Non-abstract blocks that are used for part properties and have no subtypes reference shall have a “part definition” called {block name}Part that references the definition described above.

EXAMPLE 2 Block called “StringPropertyValue” used as a part property with no subtypes has a “part definition” called “StringPropertyValuePart” that references the “StringPropertyValue” “definition”:

```
"StringPropertyValuePart": {"$ref": "#/components/schemas/StringPropertyValue"},
```

EXAMPLE 3 Block called “PropertyValue” used as a part property with one subtypes called “StringPropertyValue” has the reference to the PropertyValue “definition” (from EXAMPLE 1) and to the “part definition” for the immediate subtype “StringPropertyValuePart” (from EXAMPLE 2):

```
"PropertyValuePart": {anyOf": [
                {"$ref": "#/components/schemas/PropertyValue"},
               {"$ref": "#/components/schemas/StringPropertyValuePart"}]},
```

Abstract blocks, including select data types stereotyped to auxiliary, that are used for part properties and have subtypes shall have a “part definition” called {block name}Part that is “anyOf” the references to all the “part definition” for the immediate subtypes.

EXAMPLE 4 Abstract block called “PropertySelect” has the references to the “part definition” for all immediate subtypes, in this example “PropertyValue” and “SomethingElse”, so referencing PropertyValuePart (from EXAMPLE 3) and SomethingElsePart:

```
"PropertySelectPart": {"anyOf": [
                {"$ref": "#/components/schemas/PropertyValuePart"},
                {"$ref": "#/components/schemas/SomethingElsePart "}]},
```

These “part definition” shall be used as the reference in the part property of another block

EXAMPLE 5 a part property named “MyPropertyName” referencing a “part definition” PropertySelectPart for the block PropertySelect:

```
...
"MyPropertyName": {"items": {"$ref": "#/components/schemas/PropertySelectPart"},
                "minItems": 1,
                "type": "array"}
...
```

EXAMPLE 6 A fragment of the requestBody or responseBody for the above examples, showing an array of 3 items each of different type, and the outer required property of the name of the block indicating the different item types:

```
...
"MyPropertyName": [
  {"PropertyValue": {...}},
  {"StringPropertyValue": {...}},
  {"SomethingElse": {...}}]
...
```

5.3.7.3.3 Enumerations

Each enumeration shall have a “definition” with a field “enum” listing the enumeration literals and a “type” of the primitive type of literal.

EXAMPLE “definition” for the Enumeration called “PropertyValueCharacteristicEnum”:

```
"PropertyValueCharacteristicEnum": {
    "enum": ["upper_bound", "lower_bound", "mean", "variance", "skewness", "kurtosis",
"step_size", "delta_tolerance"],
   "type": "string"}
```

5.3.7.3.4 References

5.3.7.3.4.1 General

Blocks that are used as reference properties shall have a “reference definition” using the name of the block followed by Reference.

EXAMPLE “reference definition” for block called “**NameOfBlock**”:

```
"NameOfBlockReference": {...}
```

The following fragments of the schema show the “reference definition” in full. Shorter JSON files can be obtained by replacing the common aspects with a reference in an allOf (see B.5.4 for details).

5.3.7.3.4.2 Non-abstract block with no subtypes

Every non-abstract SysML block that has no subtypes, shall have a required property called “Reference” of “type” “object” with a definition with the following common properties:

— “refString”: required, type string and specified as an attribute in XML

```
    "refString": {"type": "string", "xml": {"attribute": true}},
```

— “reformat”: required, type string, an enumeration of “uuid”, “uri”, “address” or “unknown”, and specified as an attribute in XML

```
    "refFormat": {"enum": ["uuid", "uri", "address", "unknown"],
                  "type": "string", "xml": {"attribute": true}},
```

— “context”: optional, a “reference definition” for Organization without the “context” property

```
        "context": {
            "type": "object",
            "properties": {
                "refString": {"type": "string", "xml": {"attribute": true}},
                "refFormat": {"enum": ["uuid", "uri", "address", "unknown"],
                              "type": "string", "xml": {"attribute": true}},
                "objectType": {"enum": ["Organization"], "type": "string",
                               "xml": {"attribute": true}}
                  }
            "required": ["refString", "refFormat", "objectType"]},
```

In addition to the common properties, the definition shall have the following property specific to the block:

— "objectType: required, a string enumeration with a single choice that is the **name** of the Block, and specified as an attribute in XML

```
"objectType": {"enum": ["NameOfBlock"], "type": "string",
                       "xml": {"attribute": true}}},
```

The complete "reference definition" is shown below, where the only variable is shown in **bold text format**:

```
"NameOfBlockReference": {
  "type": "object",
  "properties": {
     "Reference": {
       "type": "object",
       "properties": {
         "refString": {"type": "string", "xml": {"attribute": true}},
         "refFormat": {"enum": ["uuid", "uri", "address", "unknown"],
                       "type": "string", "xml": {"attribute": true}},
         "context": {
           "type": "object",
           "properties": {
               "refString": {"type": "string", "xml": {"attribute": true}},
               "refFormat": {"enum": ["uuid", "uri", "address", "unknown"],
                             "type": "string", "xml": {"attribute": true}},
               "objectType": {"enum": ["Organization"], "type": "string",
                              "xml": {"attribute": true}}
                 }
           "required": ["refString", "refFormat", "objectType"]},
       "objectType": {"enum": ["NameOfBlock"], "type": "string",
                      "xml": {"attribute": true}}},
       "required": ["refString", "refFormat", "objectType"]}},
  "required": ["Reference"]}
```


5.3.7.3.4.3 Non-abstract block with subtypes

Every non-abstract SysML block with subtypes shall have a "reference definition" that is "anyOf" the "reference definition" as described for a block with no subtypes above, followed by references to the "reference definitions" of all the immediate subtypes.

EXAMPLE The block "**Organization**" has a subtype "Team", the "reference definition" for "Organization" is **anyOf** a "Reference" for "Organization" or a "Reference" for "Team" where the latter is a reference to the "reference definition" "**TeamReference**":

```
"OrganizationReference": {
  "anyOf": [
    {"type": "object",
     "properties": {
        "Reference": {"type": "object",
           "properties": {
              "refString": {"$ref": "#/components/schemas/string"},
              "refFormat": {"enum": ["uuid", "uri", "address", "unknown"],
                            "type": "string"},
              "context": {...see above...},
              "objectType": {"enum": ["Organization"]}},
           "required": ["refString", "refFormat", "objectType"]}},
     "required": ["Reference"]},
    {"$ref": "#/components/schemas/TeamReference"}]}
```

5.3.7.3.4.4 Abstract blocks

Every abstract block, including auxiliary blocks shall have a "reference definition" that is "anyOf" references to the "reference definition" of the immediate subtypes.

EXAMPLE The abstract block "ActorSelect" has subtypes of "Person" and "Organization", so the ActorSelectReference "reference definition" is anyOf "PersonReference" and "OrganziationReference":

```
"ActorSelectReference":
{"anyOf": [
      {"$ref": "#/components/schemas/PersonReference"},
      {"$ref": "#/components/schemas/OrganizationReference"}]}
```

5.3.7.3.4.5 Reference usage

The reference properties in the "definition" for the SysML Block shall reference the "reference definition" for the type of the property either directly or as an array.

EXAMPLE 1 "definition" for block "Organization" showing a reference property called "CreatedBy" of type "ActorSelect" that references "ActorItemReference", and a reference property called "InOrganization" of type "Organization" that references "OrganizationReference":

```
"Organization": {
  "properties": {
    "CreatedBy": {"$ref": "#/components/schemas/ActorSelectReference"},
    "InOrganization": {
       "items": {"$ref": "#/components/schemas/OrganizationReference"},
       "minItems": 1,
       "type": "array"},
...
```

EXAMPLE 2 A fragment of the requestBody or responseBody for EXAMPLE 1:

```
"Organization": {
  "CreatedBy": {"Reference": {"refString": "_18_4_1_aec0217_1570204365071_652837_105311
",
                              "refFormat": "uuid"
                              "objectType": "Person"}},
  "InOrganization":
            [{"Reference": {"refString": "_18_4_1_271a0567_1580742703279_359329_71983",
                              "refFormat": "uuid",
                              "objectType": "Organization"}},
             {"Reference": {"refString": "_18_4_1_1b310459_1582622003921_975386_55961",
                             "refFormat": "uuid",
                             "objectType": "Organization"}}],
  ...
```

5.3.7.3.5 Properties

5.3.7.3.5.1 General

Block properties shall be included in the "properties" field of the single required property of the name of the block, after the "$href" property. The field uses the name of the block property.

EXAMPLE The start of the JSON schema for a non-abstract block called "PropertyValue" is shown below. It has a single required property "PropertyValue" with a "properties" field, the first of which is "$href". It shows one property "MyName" with the definition hidden. The other block properties are hidden by *"...properties for the block..."*:

```
"PropertyValue": {
   "properties": {
      "PropertyValue": {
         "properties": {
            "$href": {"$ref": "#/components/schemas/uri"},
            "MyName": {...property definition...},
               ...properties for the block...
                     },
...more fields...
```

5.3.7.3.5.2 Inherited properties

There is no inheritance in JSON schema. Therefore, the properties of all super-types (including their super-types recursively) shall be included in the definition.

Inherited properties that are redefined in a subtype shall not be included.

NOTE The SysML model does not have properties with duplicate names in sub-types that are not redefined. If there are duplicate names that are not redefined, this is an error in the SysML model.

5.3.7.3.5.3 Mandatory properties

All mandatory properties are included in the "required" field. This includes mandatory properties defined in the super-types (recursive).

EXAMPLE shown in **bold text format**, the "required" field lists the members of the "properties" field that are mandatory for the block "Organization" and all its super types:

```
"Organization": {
      ...
      "properties": {...},
"required": ["CreatedBy", "CreatedOn", "Identifiers", "LastModified", "ModifiedBy"],
      "type": "object"}
...
```

5.3.7.3.5.4 Array properties

Properties with the multiplicity upper bound greater than 1 shall be defined with the field "type" set to "array" and the field "minItems" set to 1.

The field "items" for a block reference property shall reference the entity "reference definition" (see 5.3.7.3.4), for block encapsulated properties shall reference the entity "definition" (see 5.3.7.3.2 and 5.3.7.3.3).

EXAMPLE An encapsulated property called "MyPropertyValue" has a "type" of "array" with the field "items" referencing the "definition" from component/schemas and the field "minItems" set to 1:

```
"MyPropertyValue": {
       "items": {"$ref": "#/components/schemas/PropertyValue"},
       "minItems": 1,
       "type": "array"}
```

5.3.7.3.5.5 Primitive properties

It is recommended that the primitive properties are defined at the end of the components/schemas. The primitive components/schemas "definition" are shown below.

```
"boolean": { "type": "boolean"},
"dateTime": {"format": "date-time", They are referenced in the property definition in
the same manner

             "type": "string"},
"integer": {"type": "integer"},
"logical": {"enum": ["false","true","unknown" ],
             "type": "string" },
"real": {"type": "number"},
"string": {"type": "string"},
"uri": {"format": "uri",
         "type": "string"},
"ID": {"pattern": "[_A-Za-z][-_A-Za-z0-9]*",
        "type": "string"},
```

They are referenced in the property definition in the same manner as any other "definition" schema reference.

EXAMPLE Three properties that reference the primitive types.

```
"CreatedOn": {"$ref": "#/components/schemas/dateTime"},
"Text": {"$ref": "#/components/schemas/string"},
"Href": {"$ref": "#/components/schemas/uri"}
```

5.3.7.3.5.6 Inverse composite aggregation properties

The exceptions to the above rules are when an inverse composite aggregation is used. In SysML, these are displayed as an arrow with the black diamond at the same end as the arrowhead (see Figure 5). In SysML, these dictate that "1" has a reference property of type "2", and "1" is contained in "2". In STEP, they are always accompanied by one or more directed association for "relationship-like" blocks and have a multiplicity of 1.

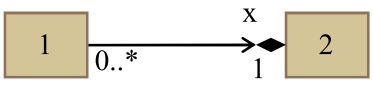

Figure 5 — Inverse composite aggregation

These shall be added as encapsulated properties to the "definition" of the "containing" block referencing the "part definition". The name of the property shall be the name of the contained block. The block that is not the "containing" block shall not include the property.

EXAMPLE 1 In Figure 6, the *InvCompTest* attribute "***Relating***" is an inverse composite aggregation, and the "***Related***" the accompanying directed association which happens to point to the same block.

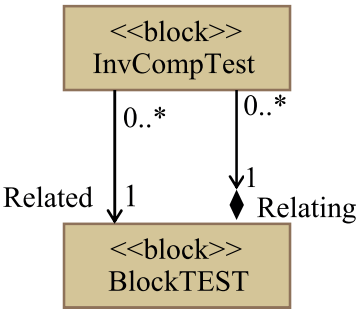

Figure 6 — Example of inverse composite aggregation

EXAMPLE 2 The resultant JSON schema for EXAMPLE 1 is shown below. The "containing" block is "BlockTEST". This has a property called "InvCompTest" that is an array where the field "items" references the "part definition" "#/components/schemas/InvCompTestPart". The "definition" for block "InvCompTest" includes a property "Related" that is a reference to the "reference definition" "#/components/schemas/BlockTESTReference". It does not include a "Relating" property.

```
"InvCompTest": {
 "properties": {
   "InvCompTest": {
     "properties": {
          ... other properties ...
          "Related": {"$ref": "#/components/schemas/BlockTESTReference"},
          ... other properties not including Relating ...

"BlockTEST": {
 "properties": {
   "BlockTEST": {
     "properties": {
          ... other properties ...
          "InvCompTest": {"items": {"$ref": "#/components/schemas/InvCompTestPart"},
                          "minItems": 1,
                          "type":"array"},
          ... other properties ...

"InvCompTestPart": {
                "anyOf": [
                    {"$ref": "#/components/schemas/InvCompTest"},
                    {"$ref": "#/components/schemas/InvCompTestRedefinedPart"}]},
```

5.3.7.3.6 Match

5.3.7.3.6.1 match_request

The "definition" match_request is referenced by the MATCH requestBody (see 5.3.6.6). It shall have two properties, "match" and "format" where only "match" is "required". Both shall reference the same block "definition" for "anyOf" the non-abstract blocks.

EXAMPLE The "match_request" "definition" showing "anyOf" for two non-abstract blocks, where the options in the "anyOf" have a required "match" property and a "format" property, both referencing the same block "definition":

```
"match_request": {
  "anyOf": [
        {"properties": {
           "match": {"$ref": "#/components/schemas/Organization"},
           "format": {"$ref": "#/components/schemas/Organization"}},
        "required": ["match"],
        "type": "object"},
        {"properties": {
            "match": {"$ref": "#/components/schemas/Person"},
            "format": {"$ref": "#/components/schemas/Person"}},
          "required": ["match"],
          "type": "object"},
        {...all other non-abstract blocks...}
        ]}
```

5.3.7.3.6.2 match_response

The “definition” match_response is referenced by the MATCH responseBody (see 5.3.6.6). It shall be an array that may be empty. The array is of one type of object where that type shall be “anyOf” the non-abstract blocks.

EXAMPLE “match_response” “definition” showing “anyOf” arrays that reference the “definition” for two non-abstract blocks:

```
"match_response": {
  "anyOf": [
    {"items": {"$ref": "#/components/schemas/Organization"},
     "minItems": 0,
     "type": "array"},
    {"items": {"$ref": "#/components/schemas/Person"},
     "minItems": 0,
     "type": "array"},
    {...other non-abstract blocks...}
    ]}
```

Annex A
(normative)

Information object registration

To provide for unambiguous identification of an information object in an open system, the following object identifier is assigned to this document:

{iso standard 10303 part(18) version(1)}

The meaning of this value is defined in ISO/IEC 8824-1, and is described in ISO 10303-1.

Annex B
(informative)

SysML to OpenAPI – Canonical XMI and equivalent in OpenAPI JSON schema

B.1 Overview conventions and assumptions.

B.1.1 General

This annex uses fragments to illustrate the transformation mapping to OpenAPI JSON Schema from the corresponding STEP data modelled in SysML, and physically stored in Canonical XMI (CXMI) file as detailed in 5.3.

It is assumed that these languages are understood so they are not explained.

B.1.2 Presentation conventions

For ease of identification, the fragments of canonical XMI (CXMI) and JSON are presented in separate boxes:

EXAMPLE 1 CXMI and JSON presented in separate boxes.

```
CXMI:

...
```

```
JSON:
...
```

The items significant to support the explanations are formatted using text effects to aid the identification of the equivalent items in the CXMI and JSON fragments. When there is more than one significant item, different text effects are used for the different items. The following text effects are used:

— **bold**;

— underlined;

— **mixed effects**.

Curly brackets "{xxx}" are used in the CXMI fragments to contain descriptive words of the content of the resulting JSON. Triple dots "…" are used to hide content not relevant to a fragment. These can contain a description of what is hidden using italic text effect.

EXAMPLE 2 Usage of **bold** and *italic* text effect to ease the identification of the significant items, and the usages of descriptive triple dots and curly brackets (only in the CXMI).

```
CXMI:

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
   <name>NameOfBlock</name>
   ...other tags...
 </packagedElement>
...other blocks...
```

JSON for tag

```
JSON:

{"name": "NameOfBlock"},
...other name ...
```

B.1.3 Common mapping conventions

B.1.3.1 Assumed sysml:Block in fragments

For all the fragments that refer to block, the following shows how a block is defined in the Canonical XMI. This is not repeated in the remaining fragments, where only `xmi:type="uml:Class"` is included and the `sysml:Block` is assumed.

Canonical XMI: Block

```
CXMI:

<sysml:Block xmi:id="{...}" xmi:uuid="{...}">
    <base_Class xmi:idref="{umlid}"/>
 </sysml:Block>

<packagedElement xmi:id="{umlid}" xmi:uuid="{...}" xmi:type="uml:Class">
...other tags...
</packagedElement>
```

B.1.3.2 Reference to external files

All the references in the SysML Canonical XMI fragments are given as `xmi:idref` that assumes the referenced element is contained in the same XMI file. When the referenced element is in a different XMI file, the `href` is used instead. This is the case for all reference to primitives and can be the case for other references.

Canonical XMI: `type href` relative reference to element with `xmi:id` "STRING" in DataTypes.xmi

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:uuid="{...}" xmi:type="uml:Property">
   <name>Text</name>
   <type href="../../../DataTypes.xmi#STRING"/>
   ...other tags...
</ownedAttribute>
```

Canonical XMI: `general href` relative reference to element in another XMI file with `xmi:id` "_18_4_1_8e001ed_1504250730055_679435_26318"

```
CXMI:

<packagedElement xmi:id="{...}" xmi:uuid="{...}" xmi:type="uml:Class">
  <name>DateTimeAssignment</name>
  <generalization xmi:id="{...}" xmi:uuid="{...}" xmi:type="uml:Generalization">
     <general href="../../Core_model/RequirementManagement/RequirementManagement.xmi#_1
8_4_1_8e001ed_1504250730055_679435_26318"/>
  </generalization>
   ...other tags...
</packagedElement >
```

B.1.3.3 Used stereotypes to represent STEP concepts

Two existing UML stereotypes are used to represent specific STEP concepts:

<<Auxiliary>> stereotypes represent select data type objects. Select data objects are represented as abstract Blocks in SysML.

<<Type>> stereotypes represent two specific types of blocks:

— blocks that represent list of lists;

— block that represents value type in order to be able to include them as member in select data type objects.

B.1.3.4 Xmi:id, xmi:uuid, and UUID

A Canonical XMI file uses `xmi:id` value to make references between all kinds of element. The value of xmi:id can be used in an `xmi:idref` attribute.

Each element in the Canonical XMI has a `xmi:uuid` (UUID). They are not used in the mapping transformations so they are not shown in the fragments.

B.2 Field "tags" for all non-abstract blocks

The following shows the Canonical XMI for a non-abstract block (shown by the absence of `<isAbstract>true</isAbstract>`) and the JSON for the tag. For each, the only variable that changes is the name of the block (see 5.3.5 for an explanation).

Canonical XMI: non-abstract block

```
CXMI:

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
   <name>NameOfBlock</name>
   ...other tags...
 </packagedElement>
```

JSON for tag

```
JSON:

{"name": "NameOfBlock"},
```

B.3 Field "paths"

B.3.1 General

The "paths" field contains the web service end points (see 5.3.6 for the definition).

B.3.2 Abstract blocks

When a block is abstract=true, there are no endpoints:

```
CXMI:

<packagedElement xmi:id="xxxIDxxx"  xmi:type="uml:Class">
    <name>NameOfAbstractBlock</name>
    <isAbstract>true</isAbstract>
   ...other tags...
</packagedElement>
```

B.3.3 Blocks that are not abstract

The following shows the Canonical XMI for a non-abstract block where it does not have `<isAbstract>true</isAbstract>`. It shows the JSON for the endpoints for GET, PUT and PATCH. For each, the only variable that changes is the name of the block (see 5.3.6 for the definition).

Canonical XMI: non-abstract block

```
CXMI:

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
  <name>NameOfBlock</name>
   ...other tags...
</packagedElement>
```

JSON schema: GET, PUT and PATCH endpoints for Blocks that are not abstract

```
JSON:

"/NameOfBlock/{uid}": {
  "get": {"description": "Returns 'NameOfBlock' objects pertaining to a uid.",
          "operationId": "get_NameOfBlock_uid",
          "parameters": [{"description": "The uid of the object to be returned.",
                          "example": "_18_4_1_1b310459_1511445569604_398628_32624",
                          "in": "path",
                          "name": "uid",
                          "required": true,
                          "schema": {"$ref": "#/components/schemas/ID" }} ],
          "responses": {"200": {"content": {
             "application/json":
                {"schema": {"$ref": "#/components/schemas/NameOfBlock"}},
             "application/xml":
                {"schema": {"$ref": "#/components/schemas/NameOfBlock"}}},
          "description": "Resources read successfully"},
                        "400": {"$ref": "#/components/responses/400"},
                        "401": {"$ref": "#/components/responses/401"},
                        "403": {"$ref": "#/components/responses/403"},
                        "404": {"$ref": "#/components/responses/404"}},
          "summary": "Return 'NameOfBlock' object by uid.",
          "tags": ["NameOfBlock"]},
  "patch": {"description": "Updates 'NameOfBlock' objects pertaining to a uid.",
            "operationId": "patch_NameOfBlock_uid",
            "parameters": [{"description": "The uid of the object to be updated.",
                            "in": "path",
                            "name": "uid",
                            "required": true,
                            "schema": {"$ref": "#/components/schemas/ID"}}],
            "requestBody": {"content": {
               "application/json-patch+json":
                   {"schema": {"type": "array", "items": {"type": "object"}}}}},
                            "required": true},
            "responses": {"200": {"$ref": "#/components/responses/200_PutPatch"},
                          "400": {"$ref": "#/components/responses/400"},
                          "401": {"$ref": "#/components/responses/401"},
                          "403": {"$ref": "#/components/responses/403"},
                          "404": {"$ref": "#/components/responses/404"}},
            "summary": "Update 'NameOfBlock' object by uid.",
            "tags": ["NameOfBlock"]},
  "put": {"description": "Replaces 'NameOfBlock' objects pertaining to a uid.",
          "operationId": "put_NameOfBlock_uid",
          "parameters": [{"description": "The uid of the object to be replaced.",
                          "in": "path",
                          "name": "uid",
                          "required": true,
                          "schema": {"$ref": "#/components/schemas/ID"}}],
          "requestBody": {"content": {
              "application/json":
                   {"schema": {"$ref": "#/components/schemas/NameOfBlock"}},
            "application/xml":
                   {"schema": {"$ref": "#/components/schemas/NameOfBlock"}}},
                            "required": true},
          "responses": {"200": {"$ref": "#/components/responses/200_PutPatch"},
                        "400": {"$ref": "#/components/responses/400"},
                        "401": {"$ref": "#/components/responses/401"},
                        "403": {"$ref": "#/components/responses/403"},
                        "404": {"$ref": "#/components/responses/404"}},
          "summary": "Replace 'NameOfBlock' object by uid.",
          "tags": ["NameOfBlock"]}},
```


B.3.4 Blocks that are not encapsulated

The following shows the Canonical XMI for a block that is encapsulated. For blocks that do not have this, the JSON for the endpoints for POST is shown (see 5.3.6.2 for the definition).

Canonical XMI: encapsulated block

```
CXMI:

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
   <name>NameOfEncapsulatedBlock</name>
   ...other tags...
</packagedElement>

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
   <name>NameOfOtherBlock</name>
   <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
      <visibility>public</visibility>
      <type xmi:idref="xxxIDxxx"/>
      <aggregation>composite</aggregation>
   </ownedAttribute>
   ...other tags...
</packagedElement>
```

JSON schema: POST endpoint for blocks that are not encapsulated

```
JSON:

"/NameOfBlock": {
  "post": {"description": "Creates new 'NameOfBlock' objects.",
           "operationId": "post_NameOfBlock",
           "requestBody": {"content": {
                 "application/json": {
                     "schema": {"$ref": "#/components/schemas/NameOfBlock"}},
                 "application/xml": {
                     "schema": {"$ref": "#/components/schemas/NameOfBlock"}}},
                          "required": true},
           "responses": {
                "201": {"$ref": "#/components/responses/201_POST"},
              "400": {"$ref": "#/components/responses/400"},
              "401": {"$ref": "#/components/responses/401"},
              "403": {"$ref": "#/components/responses/403"},
              "404": {"$ref": "#/components/responses/404"}},
           "summary": "Create a new 'NameOfBlock' object.",
           "tags": ["NameOfBlock"]}},
```

B.4 Field “components” “responses”

All but the GET 200 response are common so not dependent on the SysML canonical XMI (see 5.3.7.2 for the definition).

JSON schema: common responses

```
JSON:

"responses": {
   "200_PutPatch": {"description": "Resource updated successfully."},
   "201_POST": {"content": {
                     "application/json": {
                         "example": "_18_4_1_1b310459_1511445569604_398628_32624",
                         "schema": {"$ref": "#/components/schemas/ID"}},
                     "application/xml": {
                         "example": "_18_4_1_1b310459_1511445569604_398628_32624",
                         "schema": {"$ref": "#/components/schemas/ID"}}},
                "description": "Resource created successfully."},
  "400": {"description": "Bad Request."},
  "401": {"description": "Unauthorized." },
  "403": {"description": "Forbidden."},
  "404": {"description": "Not Found."}},
```

B.5 Field “components” “schemas”

B.5.1 General

The “components” “schemas” field contains the schemas for the blocks, enumerations and references needed by the services (see 5.3.7.3 for the definition).

B.5.2 Block schema content

B.5.2.1 General

This subclause provides fragments of JSON schema entry for SysML blocks (see 5.3.7.3.2 for the definition).

B.5.2.2 Common for non-abstract blocks

The following shows the Canonical XMI for a non-abstract block where it does not have `<isAbstract>true</isAbstract>`. It shows the JSON for the “component/schemas” field for the block. For this content fragment, the only variable that changes is the name of the block (see 5.3.7.3.2 for the definition).

Canonical XMI: non-abstract block

```
CXMI:

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
  <name>NameOfBlock</name>
   ...other tags excluding isAbstract true...
</packagedElement>
```

JSON schema: Common fragment of component > schemas for all non-abstract blocks. The properties for the block are detailed in B.5.2.3.

```
JSON:

"NameOfBlock": {
   "properties": {
      "NameOfBlock": {
         "properties": {
            "$href": {"$ref": "#/components/schemas/uri"},
                  :properties for the block...
                        },
         "required": [...],
         "type": "object"}},
    "required": ["NameOfBlock"],
    "type": "object"},
```

B.5.2.3 Properties to be included

B.5.2.3.1 General

The properties in the JSON schema shall:

— be listed alphabetically;

— include properties that are:

  — public

  — local to the block;

- local to all supertypes of the block;
- inverse composite aggregation properties.

— not include properties that are:

- visibility private;
- visibility protected;
- redefined;
- read only.

See 5.3.7.3.5 for the definition.

B.5.2.3.2 Local public properties

The following shows the Canonical XMI for a local public property where it does not have `<visibility>private</visibility>` or `<visibility>protected</visibility>`.

Canonical XMI: local public properties

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>PropertyName</name>
   ...other tags that are not <visibility>private or protected</visibility>...
</ownedAttribute>
```

JSON schema: local public properties

```
JSON:

"properties": {
 "PropertyName": {...},
... other properties ...}
```

B.5.2.3.3 Inherited properties

The following shows the fragment of Canonical XMI for a block with a super-type, followed by the fragment for that super-type block with a local public property. The fragment shows the super-type block also has a super-type, so recursion is required to locate all the inherited properties.

Inherited properties that are redefined shall not be included (see B.5.2.3.4).

NOTE All properties with duplicate names in sub-types that are redefined. If there are duplicate names that are not redefined, this is an error in the SysML model.

Canonical XMI: inherited properties

```
CXMI:

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
   <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="xxxIDxxx"/>
   </generalization>
   ... other tags ...
</packagedElement>

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
     <generalization xmi:id="{...}" xmi:type="uml:Generalization">
        <general xmi:idref="yyyIDyyy"/>
     </generalization>
       <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
       ...other tags that are not <visibility>private or protected</visibility>...
     </ownedAttribute>
   ... other tags ...
</packagedElement>
```

The JSON schema is the same as the local public properties (see B.5.2.3.2).

B.5.2.3.4 Redefined properties

Redefined properties shall not be included in the JSON. The following shows a property id=”xxxIDxxx” called “Relating” that redefines a second property id=”xxxOtherIDxxx” also called “Relating”. The second property is for a different block that is a super-type of the first block.

NOTE 1 The property that is being redefined is in a super-type block. If it is not, this is an error in the SysML model.

NOTE 2 The property that redefines another property is usually in a super-type block.

NOTE 3 The property names are usually the same but this is not mandated.

NOTE 4 Inherited properties of the same name as a local property are redefined. If they are not, this is an error in the SysML model.

Canonical XMI: redefined properties

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
  <name>Relating</name>
  <redefinedProperty xmi:idref="xxxOtherIDxxx"/>
  ... other tags ...
</ownedAttribute>

<ownedAttribute xmi:id="xxxOtherIDxxx" xmi:type="uml:Property">
  <name>Relating</name>
  ... other tags ...
</ownedAttribute>
```

B.5.2.3.5 Read only properties

Read only properties shall not be included in the JSON where the property has `<isReadOnly>true </isReadOnly>`.

Canonical XMI: read only properties

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
  ... other tags ...
  <isReadOnly>true</isReadOnly>
  ... other tags ...
</ownedAttribute>
```

B.5.2.3.6 Inverse composite aggregation properties

Figure B.1 shows the block definition diagram for a named inverse composite aggregation property used in this fragment.

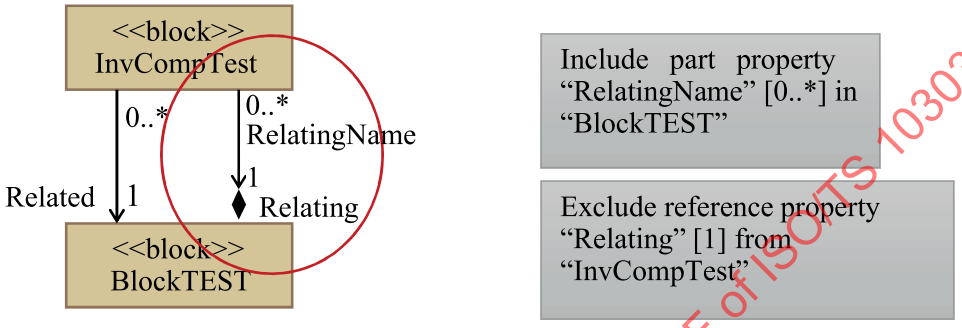

Figure B.1 — Block definition diagram of inverse composite aggregation

The following shows how to identify if a property of a block is an inverse composite aggregation and so shall be excluded.

An `ownedAttribute` has a referenced `association` that has an `ownedEnd` (of the same `type` as the block) that has an `aggregation composite`.

Canonical XMI: inverse composite aggregation properties to be excluded

```
CXMI:

<packagedElement xmi:id="xxxID_B1_xxx" xmi:type="uml:Class">
  <name>InvCompTest</name>
  <ownedAttribute xmi:id="xxxID_P_xxx" xmi:type="uml:Property">
    <name>Relating</name>
    <type xmi:idref="xxxID_B2_xxx"/>
    <association xmi:idref="xxxID_A_xxx"/>
    ...other tags that are not <visibility>private or protected</visibility>...
  </ownedAttribute>
  ...other tags...
</packagedElement>


<packagedElement xmi:id="xxxID_A_xxx" xmi:type="uml:Association">
   <memberEnd xmi:idref="xxxID_P_xxx"/>
   <memberEnd xmi:idref="xxxID_OE_xxx"/>
   <ownedEnd xmi:id="xxxID_OE_xxx" xmi:type="uml:Property">
     <type xmi:idref="xxxID_B1_xxx"/>
     <aggregation>composite</aggregation>
    <association xmi:idref="xxxID_A_xxx"/>
     ...other tags that are not <visibility>private or protected</visibility>...
   </ownedEnd>
</packagedElement>

<packagedElement xmi:id="xxxID_B2_xxx" xmi:type="uml:Class">
   <name>BlockTest</name>
   ...other tags...
</packagedElement>
```

The following shows how to identify if there is an inverse composite aggregation property that shall be included for the block:

— an `ownedEnd` of `xmi:type="uml:Property"` has an `aggregation composite` and does not have `visibility private` or `protected` indicates there is an inverse composite aggregation (**bold** text effect);

— the block for which the property is to be included is the `type` of the `ownedAttribute` of `xmi:type Property` that references the same association (underline text effect);

— if the `ownedEnd` has a `name` (for the fragments this is “RelatingName”), then this shall be the name of the property, otherwise name of the property is `name` of the block that is the type of the `ownedEnd` (**bold and underline** text effect).

Canonical XMI: inverse composite aggregation properties to be included

```
CXMI:

<packagedElement xmi:id="xxxID_A_xxx" xmi:type="uml:Association">
   <memberEnd xmi:idref="xxxID_P_xxx"/>
   <memberEnd xmi:idref="xxxID_OE_xxx"/>
   <ownedEnd xmi:id="xxxID_OE_xxx" xmi:type="uml:Property">
     <name>RelatingName</name>
     <type xmi:idref="xxxID_B1_xxx"/>
     <aggregation>composite</aggregation>
     <association xmi:idref="xxxID_A_xxx"/>
     ...other tags that are not <visibility>private or protected</visibility>...
   </ownedEnd>
</packagedElement>

<packagedElement xmi:id="xxxID_B2_xxx" xmi:type="uml:Class">
   <name>BlockTest</name>
   ...other tags...
</packagedElement>

<packagedElement xmi:id="xxxID_B1_xxx" xmi:type="uml:Class">
  <name>InvCompTest</name>
  <ownedAttribute xmi:id="xxxID_P_xxx" xmi:type="uml:Property">
    <name>Relating</name>
    <type xmi:idref="xxxID_B2_xxx"/>
    <association xmi:idref="xxxID_A_xxx"/>
    ...other tags that are not <visibility>private or protected</visibility>...
  </ownedAttribute>
  ...other tags...
</packagedElement>
```

In the following fragments, the property is explained in more detail in B.5.2.4. However, the content of the "$ref" is always an xxxPart because it is a composite property.

JSON schema: inverse composite aggregation named property

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
         ...other properties...
          "RelatingName": {
              "items": {"$ref": "#/components/schemas/InvCompTestPart"},
              "minItems": 1,
              "type": "array"},
```

JSON schema: inverse composite aggregation unnamed property

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
         ...other properties...
          "InvCompTest": {
              "items": {"$ref": "#/components/schemas/InvCompTestPart"},
              "minItems": 1,
              "type": "array"},
```

B.5.2.4 Property content

B.5.2.4.1 General

Subclauses B.5.2.4.2 to B.5.2.4.5 explain how to handle properties that are:

— individuals or arrays;

— primitives, parts or references.

B.5.2.4.2 Understanding multiplicity in the SysML Canonical XMI

In the SysML canonical XMI, the multiplicity is specified with the tags `lowerValue` and `upperValue`. These can be contained in `ownedAttribute` or `ownedEnd`. The types of multiplicity used are `[1]`, `[0..1]`, `[0..*]`, `[1..*]`, `[n..*]`, `[n..m]`.

The Canonical XMI rules are as follows.

— When there is no `lowerValue` specified the multiplicity lower value is 1.

— When there is a `lowerValue`, but no inner `value` tag, the multiplicity lower value is 0.

— When there is a `lowerValue`, and an inner `value` tag, the multiplicity lower value is the content of the `value` tag.

— When there is no `upperValue` specified the multiplicity upper value is 1.

— When there is an upper `upperValue` there shall also be an inner `value` tag, and the multiplicity upper value is the content of the `value` tag.

Canonical XMI: property with multiplicity of `[1]`

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>{...}</name>
   ...other tags that are not <visibility>private or protected</visibility>...
   ...other tags not <lowerValue></lowerValue> or <upperValue></upperValue>...
</ownedAttribute>
```

Canonical XMI: property with multiplicity of `[0..1]`

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>{...}</name>
   ...other tags that are not <visibility>private or protected</visibility>...
   <lowerValue xmi:id="{...}" xmi:type="{...}"/>
   ...other tags not <upperValue></upperValue>...
</ownedAttribute>
```

Canonical XMI: property with multiplicity of `[0..*]`

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>{...}</name>
   ...other tags that are not <visibility>private or protected</visibility>...
   <lowerValue xmi:id="{...}" xmi:type="{...}"/>
   <upperValue xmi:id="{...}" xmi:type="{...}">
     <value>*</value>
    </upperValue>
   ...other tags...
</ownedAttribute>
```

Canonical XMI: property with multiplicity of `[1..*]`

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>{...}</name>
   ...other tags that are not <visibility>private or protected</visibility>...
   ... other tags not <lowerValue></lowerValue>...
   <upperValue xmi:id="{...}" xmi:type="{...}">
     <value>*</value>
    </upperValue>
   ...other tags...
</ownedAttribute>
```

Canonical XMI: property with multiplicity of `[n..*]` where $n = 2$

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>{...}</name>
   ...other tags that are not <visibility>private or protected</visibility>...
    <lowerValue xmi:id="{...}" xmi:type="{...}">
     <value>2</value>
    </lowerValue>
   <upperValue xmi:id="{...}" xmi:type="{...}">
     <value>*</value>
    </upperValue>
   ...other tags...
</ownedAttribute>
```

Canonical XMI: property with multiplicity of `[n..m]` where $n = 2$ and $m = 3$

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>{...}</name>
   ...other tags that are not <visibility>private or protected</visibility>...
    <lowerValue xmi:id="{...}" xmi:type="{...}">
     <value>2</value>
    </lowerValue>
   <upperValue xmi:id="{...}" xmi:type="{...}">
     <value>3</value>
    </upperValue>
   ...other tags...
</ownedAttribute>
```

B.5.2.4.3 Individual properties

A single value is when the upper value of the multiplicity is 1 (see Understanding multiplicity in B.5.2.4.2). The control of whether this is a mandatory property is governed by the field "`required` ": (see B.5.2.5).

NOTE The content of `$ref` in the JSON schema fragment is explained in B.5.2.4.5.

Canonical XMI: property with multiplicity `[1]`

```
CXMI:

<packagedElement xmi:id="{...}" "xmi:type="uml:Class">
  <name>BlockTEST</name>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
    <name>SingleProperty</name>
    ...other tags not <lowerValue></lowerValue> or <upperValue></upperValue>...
  </ownedAttribute>
</packagedElement>
```

Canonical XMI: property with multiplicity `[0..1]`

```
CXMI:

<packagedElement xmi:id="{...}" "xmi:type="uml:Class">
  <name>BlockTEST</name>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
    <name>SingleProperty</name>
    <lowerValue xmi:id="{...}" xmi:type="{...}"/>
    ...other tags not <upperValue></upperValue>...
  </ownedAttribute>
</packagedElement>
```

JSON schema: Property with multiplicity of `[1]` or `[0..1]`

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
          ...other properties...
          "SingleProperty": {"$ref": "#/components/schemas/..."},
      ...other properties...
```

B.5.2.4.4 Array properties

An array is when the upper value of the multiplicity is greater than 1 or * (see Understanding multiplicity in B.5.2.4.2).

NOTE 1 In JSON schema, an array is “items”: of “type”: “array” with “minItems”: and “maxItems”: as needed.

The value of "`minItems`": shall be at least 1 so the array always has content. The control of whether there is an array or not is governed by the field "`required` ": (see B.5.2.5).

NOTE 2 The content of `$ref` in the JSON schema fragment is explained in B.5.2.4.5.

Canonical XMI: property with multiplicity `[0..*]`

```
CXMI:

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <name>BlockTEST</name>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
    <name>Context</name>
    ...other tags...
    <lowerValue xmi:id="{...}" xmi:type="{...}"/>
    <upperValue xmi:id="{...}" xmi:type="{...}">
   <value>*</value>
    </upperValue>
    ...other tags...
  </ownedAttribute>
</packagedElement>
```

JSON schema: All required properties are listed in the field “required” after the “properties” field.

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
          ...other properties...
          "Context": {"items": {
                       "$ref": "#/components/schemas/..."},
                       "minItems": 1,
                       "type": "array"
                      },
           ...other properties...
```

Canonical XMI: property with multiplicity `[2..3]`

```
CXMI:

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <name>BlockTEST</name>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
    <name>ArrayProperty</name>
    ...other tags...
    <lowerValue xmi:id="{...}" xmi:type="{...}">
<value>2</value>
    </lowerValue>
    <upperValue xmi:id="{...}" xmi:type="{...}">
     <value>3</value>
    </upperValue>
    ...other tags...
  </ownedAttribute>
</packagedElement>
```

JSON schema: All required properties are listed in the field “required” after the “properties” field.

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
          ...other properties...
          "ArrayProperty": {"items": {
                       "$ref": "#/components/schemas/..."},
                       "minItems": 2,
                       "maxItems": 3,
                       "type": "array"
                      },
          ...other properties...
```

B.5.2.4.5 $ref schema reference for primitive, part and reference properties

B.5.2.4.5.1 General

Subclauses B.5.2.4.5.1 to B.5.2.4.5.4 explain how the $ref schema reference for primitive properties, part properties and reference properties shall be used.

B.5.2.4.5.2 Primitive schema $ref

A primitive is identified by the `xmi:type="uml:PrimitiveType"` attribute. All common the primitives are stored in the DataTypes.xmi, so any `href` that contains this file shall be assumed to be a primitive. In addition, the `xmi:id` for the primitives in DataTypes.xmi can be assumed to be the capitalized form of the `name`. All `packagedElements` in DataTypes.xmi are treated as primitives (some are `xmi:type="uml:Enumeration"`), therefore the portion of the href after the # can be used to identify the JSON schema primitive to be used.

NOTE All primitive properties in the SysML canonical XMI include the tags `<aggregation>composite</aggregation>`. If they do not include this tag, this is an error in the SysML model.

Canonical XMI for a primitive in DataTypes.xmi

```
CXMI:

<packagedElement xmi:id="STRING" xmi:type="uml:PrimitiveType">
   <name>String</name>
</packagedElement>
```

Canonical XMI: `type href` relative reference to string primitive in DataTypes.xmi

```
CXMI:

<ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
   <name>Text</name>
   <type href="../../../DataTypes.xmi#STRING"/>
   ...other tags...
</ownedAttribute>
```

JSON schema: property with string primitive

```
JSON:

"properties": {
          ...other properties...
         "Text": {"$ref": "#/components/schemas/string"}
         ...other properties...
```

Table B.1 shows the xmi:id and containing file for the primitives in the SysML XMI that have an equivalent JSON schema primitive. See B.5.6 for how to construct the JSON schema primitive schemas.

Table B.1 — SysML Canonical XMI primitives and equivalent in JSON schemaSysML Canonical XMI primitives and equivalent in JSON schema

XMI File	xmi:id	JSON
Core_model/CommonRessources/CommonRessources	_DateTimeString	dateTime
Core_model/CommonRessources/CommonRessources	_Uri	uri
DataTypes.xmi	INTEGER	integer
DataTypes.xmi	STRING	string
DataTypes.xmi	BOOLEAN	boolean
DataTypes.xmi	LOGICAL	logical
DataTypes.xmi	REAL	real

B.5.2.4.5.3 Part schema $ref

A part property in the SysML Canonical XMI is where an `ownedAttribute` includes the tag `<aggregation>composite</aggregation>`. The other case is for inverse composite aggregation properties (see B.5.2.3.6). The name of the `$ref` schema is the `name` of the block (underline text effect) that is the `type` (**bold and underline** text effect) of the `ownedAttribute`, appended with the string with "Part" (**bold** text effect).

NOTE 1 The construction of "Part" schemas is explained in B.5.3.

NOTE 2 The construction of single properties is explained in B.5.2.4.3 and array properties is explained in B.5.2.4.4.

NOTE 3 This is the same whether the type of the part is a block (including abstract and auxiliary blocks) or an enumeration. The exception is when the part is a primitive (see B.5.2.4.5.2).

Canonical XMI: Part property that is an array

```
CXMI:

<packagedElement xmi:id="{...}" "xmi:type="uml:Class">
  <name>BlockTEST</name>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
    <name>Context</name>
    ...other tags...
    <type xmi:idref="xxxIDxxx"/>
    <aggregation>composite</aggregation>
    ...other tags...
  </ownedAttribute>
</packagedElement>

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
   <name>AssumptionContextItem</name>
   ...other tags...
</packagedElement>
```

JSON schema: $ref schema for a part property that is an array

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
          ...other properties...
          "Context": {"items": {
                       "$ref": "#/components/schemas/AssumptionContextItemPart"},
                       "minItems": 1,
                       "type": "array"
                      },
           ...other properties...
```

B.5.2.4.5.4 Reference schema $ref

A reference property in the SysML Canonical XMI is where an `ownedAttribute` does not include the tag `<aggregation>composite</aggregation>`. The name of the `$ref` schema is the `name` of the block (underline text effect) that is the `type` (**bold and underline** text effect) of the `ownedAttribute`, appended with the string with "Reference" (**bold** text effect).

NOTE 1 The construction of "Reference" schemas is explained B.5.4.

NOTE 2 The construction of single properties is explained in B.5.2.4.3 and array properties is explained in B.5.2.4.4.

NOTE 3 This is the same whether the type of the reference is a block (including abstract and auxiliary blocks). The SysML model does not contain any enumeration or primitive reference properties. If there are, this is an error in the SysML model.

Canonical XMI: Part property that is an array

```
CXMI:

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <name>BlockTEST</name>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
    <name>Assumes</name>
    ...other tags...
    <type xmi:idref="xxxIDxxx"/>
    ...other tags excluding <aggregation>composite</aggregation>...
  </ownedAttribute>
</packagedElement>

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
   <name>AssumedItem</name>
   ...other tags...
</packagedElement>
```

JSON schema: $ref schema for a part property that is an array

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
          ...other properties...
          "Assumes": {"items": {
                       "$ref": "#/components/schemas/AssumedItemReference"},
                       "minItems": 1,
                       "type": "array"
                      },
           ...other properties...
```

B.5.2.5 Required properties

The required properties are those where the lower value of the multiplicity is 1 (see B.5.2.4.2). This occurs either when there is no `<lowerValue>` (such as default of 1) or when there is a `<lowerValue><value>` (when no `<value>` specified the default is 0).

Canonical XMI: required property "Assumes" and optional "Context"

```
CXMI:

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <name>BlockTEST</name>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
    <name>Context</name>
    ...other tags...
    <lowerValue xmi:id="{...}" xmi:type="{...}"/>
    <upperValue xmi:id="{...}" xmi:type="{...}">
      <value>*</value>
    </upperValue>
    ...other tags...
  </ownedAttribute>
  <ownedAttribute xmi:id="{...}" xmi:type="uml:Property">
     <name>Assumes</name>
     ... other tags not <lowerValue></lowerValue>...
     <upperValue xmi:id="{...}" xmi:type="{...}">
        <value>*</value>
     </upperValue>
     ...other tags...
   </ownedAttribute>
 </packagedElement>
```

JSON schema: All required properties are listed in the field "required" after the `"properties"` field

```
JSON:

"BlockTEST": {
  "properties": {
    "BlockTEST": {
      "properties": {
          "Assumes": {...},
          "Context": {...},
          ... more (inherited) properties...
       },
       "required": ["Assumes", "...more (inherited) required properties..."],
```

B.5.3 BlockPart

B.5.3.1 General

Any SysML block that is used in a part property (or value property) shall have a JSON "Part" schema (see 5.3.7.3.2 for the definition).

NOTE 1 A SysML block is not used as the type for both part properties and reference properties. If there are cases, then this is an error in the SysML model. N326_Canonical_xmi_example_input.xmi listed in Annex C includes examples of this error.

B.5.3.2 Abstract blocks including auxiliary (SELECT)

All SysML blocks that have the tag `<isAbstract>true</isAbstract>` are handled the same in the JSON schema whether they are stereotyped to `StandardProfile:Auxiliary` or not. The JSON schema is `anyOf` all the subtype block part schemas. This embeds recursion through subtypes of subtypes.

The embedded recursion is illustrated in the following fragment where "AssumptionContextItem" has an abstract subtype "VersionableObject" which in turn has a subtype "BlockTEST". When all the

`anyOf` references are resolved the final list only contains the schemas for the non-abstract blocks. The resolved `anyOf` for "AssumptionContextItem" contains "TeamTEST" and "BlockTEST".

NOTE This is the same approach as used for BlockReference (see B.5.4.2) except appending "Part" instead of "Reference" in the $ref.

Canonical XMI: abstract auxiliary and abstract blocks for parts

```
CXMI:

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
   <name>AssumptionContextItem</name>
   <isAbstract>true</isAbstract>
     ...other tags...
</packagedElement>

<StandardProfile:Auxiliary xmi:id="{...}">
    <base_Class xmi:idref="xxxIDxxx"/>
 </StandardProfile:Auxiliary>

... following object is subtype of AssumptionContextItem and is not abstract...
<packagedElement xmi:id="xxxID_TTxxx" xmi:type="uml:Class">
  <name>TeamTEST</name>
   <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="xxxIDxxx"/>
   </generalization>
     ...other tags...
</packagedElement>

... following abstract object is subtype of AssumptionContextItem...
<packagedElement xmi:id="xxxID_VOxxx" xmi:type="uml:Class">
  <name>VersionableObject</name>
   <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="xxxIDxxx"/>
   </generalization>
   <isAbstract>true</isAbstract>
     ...other tags...
</packagedElement>

... following abstract object is subtype of AssumptionContextItem...
<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <name>BlockTEST</name>
   <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="xxxID_VOxxx"/>
   </generalization>
     ...other tags...
</packagedElement>
```

JSON schema: abstract auxiliary and abstract blocks for parts

```
JSON:

"AssumptionContextItemPart": {
   "anyOf": [{"$ref": "#/components/schemas/TeamTESTPart"},
             {"$ref": "#/components/schemas/VersionableObjectPart"}]
           },

"VersionableObjectPart": {
   "anyOf": [{"$ref": "#/components/schemas/BlockTESTPart"}]},
```

B.5.3.3 Blocks with no subtypes

A block that is not abstract and has no subtypes is when there are no cases of `<general xmi:idref=""/>` equal to the `xmi:id` of the block in the Canonical XMI. The JSON schema is a reference to the Block schema (see 5.3.7.3.2 for the definition).

NOTE This seeming repetition is to allow for consistent of block references and to enable recursion.

Canonical XMI: block that are used as parts and have no subtypes

```
CXMI:

<packagedElement xmi:id="xxxID_TTxxx" xmi:type="uml:Class">
  <name>TeamTEST</name>
   ...other tags...
</packagedElement>

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="...no cases of xxxID_TTxxx..."/>
   </generalization>
     ...other tags...
</packagedElement>
```

JSON schema: blocks that are used for parts and have no subtypes

```
JSON:

"TeamTESTPart": {"$ref": "#/components/schemas/TeamTEST"},
```

B.5.3.4 Blocks with subtypes

The non-abstract blocks with subtypes combine the approaches of the abstract blocks (see B.5.3.2) and the blocks with no subtypes (see B.5.3.3).

The JSON schema shall be `anyOf` reference to the block schema (see B.5.2), and all the subtype block "Part" schemas. This embeds recursion through subtypes of subtypes.

Canonical XMI:

```
CXMI:

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
  <name>InvCompTest</name>
...other tags...
</packagedElement>

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <name>InvCompTestRedefined</name>
  <generalization xmi:id="{...}" xmi:type="uml:Generalization">
    <general xmi:idref="xxxIDxxx"/>
  </generalization>
...other tags...
</packagedElement>
```

JSON schema:

```
JSON:

"InvCompTestPart": {
    "anyOf": [{"$ref": "#/components/schemas/InvCompTest"},
              {"$ref": "#/components/schemas/InvCompTestRedefinedPart"}]
           },
```

B.5.4 BlockReference

B.5.4.1 General

Any SysML slock that is used in a reference property shall have a JSON "Reference" schema (see 5.3.7.3.4 for the definition).

NOTE A SysML block is not used as the type for both part properties and reference properties. If there are cases of this in the canonical XMI, then this is an error in the SysML model. N326_Canonical_xmi_example_input.xmi listed in Annex C includes examples of this error.

B.5.4.2 Abstract blocks including auxiliary (SELECT)

All SysML blocks that have the tag `<isAbstract>true</isAbstract>` shall be handled the same in the JSON schema whether they are stereotyped to `StandardProfile:Auxiliary` or not. The JSON schema shall be `anyOf` all the subtype block "Reference" schemas. This embeds recursion through subtypes of subtypes.

The embedded recursion is illustrated in the following fragment where "ActorItem" has subtypes of "Organization" and "Person", but "Organization" has a subtype "TeamTEST" (see B.5.4.4). When all the `anyOf` references are resolved, the final list contains the schemas for the non-abstract blocks "References". Therefore, the resolved "ActorItem" is `anyOf` Person, Organization and TeamTEST.

NOTE This is the same approach as used for BlockPart (see B.5.3.2) except appending "Reference" instead of "Part" in the `$ref`.

Canonical XMI: abstract auxiliary and abstract blocks for references

```
CXMI:

<packagedElement xmi:id="xxxIDxxx" xmi:type="uml:Class">
   <name>ActorItem</name>
   <isAbstract>true</isAbstract>
     ...other tags...
</packagedElement>

<StandardProfile:Auxiliary xmi:id="{...}">
    <base_Class xmi:idref="xxxIDxxx"/>
 </StandardProfile:Auxiliary>

... following object is subtype of ActorItem and is not abstract...
<packagedElement xmi:id="xxxID_Oxxx" xmi:type="uml:Class">
  <name>Organization</name>
   <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="xxxIDxxx"/>
   </generalization>
  ...other tags...
</packagedElement>

... following object is subtype of ActorItem and is not abstract...
<packagedElement xmi:id="xxxID_Pxxx" xmi:type="uml:Class">
  <name>Person</name>
   <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="xxxIDxxx"/>
   </generalization>
  ...other tags...
</packagedElement>
```

JSON schema: abstract auxiliary and abstract blocks for references

```
JSON:

"ActorItemReference": {
    "anyOf": [{"$ref": "#/components/schemas/OrganizationReference"},
              {"$ref": "#/components/schemas/PersonReference"}]
            },
```

B.5.4.3 Non-abstract blocks with no subtypes

A block that is not abstract and has no subtypes is when there are no cases of `<general xmi:idref=""/>` equal to the `xmi:id` of the block.

The BlockReference in the JSON schema has properties that are common to all BlockReferences. These may be held in a single common schema called `commonRef`. There shall be a property called `objectType` which is fixed for the block. All the properties when used in the XML payload are treated as xml attributes for consistency with the XSD (see 5.3.7.3.4.2 for the definition).

Canonical XMI:

```
CXMI:

<packagedElement xmi:id="xxxID_TTxxx" xmi:type="uml:Class">
  <name>TeamTEST</name>
   <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="xxxID_Oxxx"/>
   </generalization>
     ...other tags...
</packagedElement>

<packagedElement xmi:id="{...}" xmi:type="uml:Class">
  <generalization xmi:id="{...}" xmi:type="uml:Generalization">
     <general xmi:idref="...no cases of xxxID_TTxxx..."/>
   </generalization>
     ...other tags...
</packagedElement>
```

JSON schema: blocks for references that have no subtypes when a single common schema is used

```
JSON:

"TeamTESTReference": {
   "properties": {
      "Reference": {
          "allOf": [{"$ref": "#/components/schemas/commonRef"},
                    {"properties": {"objectType": {"enum": ["TeamTEST"],
                                                   "type": "string",
                                                   "xml": {"attribute": true}}},
                     "required": ["objectType"],
                     "type": "object"}]}},
                "required": ["Reference"],
                "type": "object"
            },
```

JSON schema: common properties for all reference schemas in a single common schema